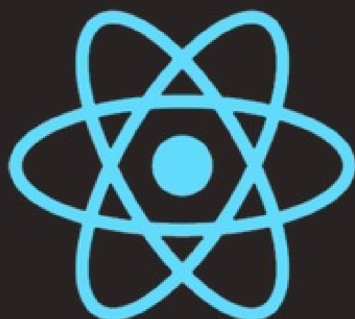
目錄

从零开始学 ReactJS（ReactJS 101）

一本给初学者的 React 中文入门教学书，由浅入深学习 ReactJS 生态系 (Flux, Redux, React Router, ImmutableJS, React Native, Relay/GraphQL etc.)，打造跨平台应用程序。



從零開始學 ReactJS

由淺入深學習 React 生態系

KD Chang
張凱迪 著

相关连结（Links）

1. 从零开始学 ReactJS（ReactJS 101）粉丝页
2. 繁体中文范例程序码和书籍内容连载地址

3. 勘误、许愿、建议或提問

翻译版本（Translate）

1. 简体中文版本 by @carlleton
2. 前端圈简体中文版本 by @blueflylin 特别感谢前端圈小夥伴！

若需翻译成其他语言版本，請先 `fork` 一份 `repo` 到自己的 Guthub 并另外开新的 `branch` 。最后将翻译版本连接更新在 `master` 分支中 `README.md` 的 `相关连接（Links）` 后发送 `Pull Request` ，谢谢您。

目录（Table of Contents）

先备知识（Prior Knowledge）

本书针对已具备基本 HTML、CSS 和 JavaScript 和 DOM 操作知识的读者设计，但若读者对上述的技术仍不熟悉的话，建议可以先行参考：MDN、Codecademy、W3C School、JavaScript核心 或是参考笔者 之前的教学讲义 进行学习。另外，本书全书范例都将以 ES6+ 撰写，若需参考 ES5 用法，请参考附录一的 React ES5、ES6+ 常见用法对照表。

关于作者（Author）

@kdchang 文艺型开发者，梦想是做出人们想用的产品和办一所心目中理想的学校，目前专注在 Mobile 和 IoT 应用开发。A Starter & Maker. JavaScript, Python & Arduino/Android lover.:)

版权许可（License）

关键字（Keywords）

React, React Native, React Router, Flux, Redux, Node, Express, ImmutableJS, NPM, Babel, Browserify, Webpack, Gulp, Grunt, Pure Functions, PropTypes, Stateless Functional Components, Presentational Components, ES6, ES5, JSX, Jest, Unit Test, Component, Relay, GraphQL, Universal/Isomorphic, React Tutorial React教程, React教学, 学React, React Tutorial, Tutorial, Ecosystem, Front-End

Ch01 前端工程简介和 React 生态系简介

:door: 任意门

| 回首页 |

Web 前端工程入门简介



前言

随著现代化网页（Modern Web）开发专业和复杂性的提升以及对用户体验越来越高的要求下，网页开发已从过去的 Web Develpoer 一夫当关，转向专业分工，更加细分成网页前端（Web Front End）、网页后端（Web Back End）等职位。此外，由于跨平台、跨浏览器的需求日益增加，技术变化更迭快速，市场上对于前端工程师（Web Front End Engineer）的需求也与日俱增，前端工程的（Front End Engineering）所要面对的挑战也越来越多。



前端工程范畴

事实上，在目前的业界，前端工程的定位光谱非常广泛，有聚焦在网页设计（Web Design），也有专注在软件工程（Software Engineering）的部份，本书则是将前端工程定位在软件工程的范畴。而 HTML、CSS 和 JavaScript 是前端工程最重要的技术基础。过去一段时间，我们所认为的前端工程主要专注在浏览器平台，但现在的 Web 平台已经不再局限于桌面浏览器，而是必须面对更多的跨平台、跨浏览器的应用开发场景，其中包含：

1. 网页浏览器（Web Browser），一般的网页应用程序开发
2. 通过 CLI 指令去操作的 Headless 浏览器（Headless Application）。例如：phantomJS、CasperJS 等
3. 运作在 WebView 浏览器核心（WebView Application）的应用。例如：Apache Cordova、Electron、NW.js 等行动、桌面应用程序开发
4. 原生应用程序（Native Application），通过 Web 技术撰写原生应用程序。例如：React Native、Native Script 等

过去几年，前端开发就像经历了文艺复兴（Rinascimento）的年代，开始了各种框架、套件百花齐放的时代。虽然现在有更多好用工具可以协助开发，但前端工程师似乎并没有变得比较轻松。以往若能妥善运用 jQuery 等函数库就可以应付大部分前端工程师的工作，但现在前端招聘广告上不仅要求精通 HTML、CSS 和 JavaScript，还要对于还要对于 Backbone、Ember、Angular、React、Vue 等 JavaScript 框架或函数库有一定程度的了解。

在众多 JavaScript 框架或函数库中，React 是 Facebook 推出的开源 JavaScript Library，它的出现让许多革新性的 Web 观念开始流行起来，例如：Virtual DOM、Web Component、更直觉的定义式 UI 设计、更优雅地实现 Server Rendering 等。接下来本书将通过介绍 React 生态系（ecosystem）带领读者入门 React 的世界，让读者可以从零开始真的动手用 React 开发跨平台应用程序。

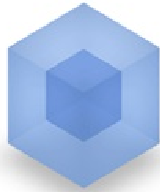
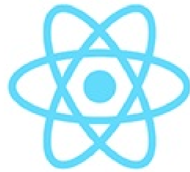
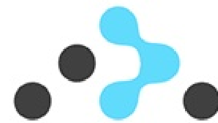
（image via bsdacademy、firebase）

:door: 任意门

| 回首页 | 下一章：React 生态系（Ecosystem）入门简介 |

| 纠错、提问或许愿 |

React 生态系（Ecosystem）入门简介

根据 React 官方网站 的说明：React 是一个专注于 UI（View）的 JavaScript 函数库（Library）。自从 Facebook 于 2013 年开源 React 这个函数库后，相关的生态系开始蓬勃发展。事实上，通过学习 React 生态系（ecosystem）的过程中，可以让我们顺便学习现代化 Web 开发的重要观念（例如：模块化、ES6+、Webpack、Babel、ESLint、函数式程序设计等），成为更好的开发者。

ReactJS

ReactJS 是 Facebook 推出的 JavaScript 函数库，若以 MVC 框架来看，React 定位是在 View 的范畴。在 ReactJS 0.14 版之后，ReactJS 更把原先处理 DOM 的部分独立出去（react-dom），让 ReactJS 核心更单纯，也更符合 React 所倡导的 `Learn once, write everywhere` 的理念。事实上，ReactJS 本身的 API 相对单纯，但由于整个生态系非常庞大，因此学习 React 却是一条漫长的道路。此外，当你想把 React 应用在你的应用程序时，你通常必须学习整个 React Stack 才能充分发挥 React 的最大优势。

JSX

事实上，JSX 并非一种全新的语言，而是一种语法糖（Syntatic Sugar），一种语法类似 XML 的 ECMAScript 语法扩充。在 JSX 中 HTML 和组建这些元素标签的程序码有紧密的关系，这和过去我们强调 HTML、JavaScript 分离的观念有很大不同。当然，你可以选择不要在 React 使用 JSX，不过相信我，当你真正开始编写 React 元件（Component）时，你会很庆幸有 JSX 真好。

NPM

NPM（Node Package Manager）是 Node.js 下的主流包管理工具。在 NPM 上有非常多的包，可以让你不用再重造轮子，更可以让你可以轻松用指令管理不同的包。由于 NPM 主要是基于 CommonJS 的规范，通常必须搭配 Browserify 这样的工具才能在前端使用 NPM 的模块。然而因 NPM 是基于 Nested Dependency Tree，不同的包有可能会在引入依赖时会引入相同但不同版本的包，造成档案大小过大的情形。这和另一个包管理工具 Bower 专注在前端包且使用 Flat Dependency Tree（让使用者决定相依的包版本）是比较不同的地方。

ES6+

ES6+ 系指 ES6（ES2015）和 ES7 的联集，在 ES6+ 新的标准当中引入许多新的特性和功能，弥补了过去 JavaScript 被诟病的一些特性。由于未来 React 将以支持 ES6+ 为主，因此直接学习 ES6+ 用法是相对好的选择，本书的所有范例也将会以 ES6+ 编写。

Babel

由于并非所有浏览器都支持 ES6+ 语法，所以通过 Babel 这个 JavaScript 编译器（可以想成是翻译机）可以让你的 ES6+ 、JSX 等程序码转换成浏览器可以看的懂得语法。通常会在数据夹的 root 位置加入 `.babelrc` 进行转译规则 `preset` 和引用外挂（plugin）的设定。

JavaScript 模块化开发

随著 Web 应用程序的复杂性提高，JavaScript 模块化开发已经成为必然的趋势，以下简单介绍 JavaScript 模块化的相关规范。事实上，在一开始没有官方定义的标准时出现了各种社群自行定义的规范和实践。

1. CDN-Based

   也就是最传统的 `<script>` 引入方式，然而使用这种方式虽然简单方便，但在开发实际中大型应用程序时会产生许多弊端：

   - 全局作用域容易造成变数污染和冲突

- 文件只能依照 `<script>` 顺序载入，不具弹性
- 在大型专案中各种资源和版本难以维护
- 必须由开发者自行判断模块和函数库之间的依赖关系

2. AMD

   Asynchronous Module Definition 简称 AMD，为非同步载入模块的规范，其在定义时模块时即需定义依赖的模块。AMD 常用于浏览器端，其最著名的实践为 RequireJS

   基本格式：

   ```
   define(id?, dependencies?, factory);
   ```

3. CommonJS

   CommonJS 规范是一种同步模块载入的规范。以 Node.js 其遵守 CommonJS 规范，使用 `require` 进行模块同步载入，并通过 `exports` 、 `module.exports` 来输出模块。主要实现为 Node.js 服务器端的同步载入和浏览器端的 Browserify。

4. CMD

   CMD 全称为 Common Module Definition，其规范和 AMD 类似，但相对简洁，却又保持和 CommonJS 的兼容性。其最大特色为：依赖就近，延迟执行。主要实现为：Sea.js。

5. UMD

   Universal Module Definition 是为了要兼容 CommonJS 和 AMD 所设计的规范，希望让模块能跨平台执行。

6. ES6 Module

   ECMAScript6 的标准中定义了 JavaScript 的模块化方式，让 JavaScript 在开发大型复杂应用程序时上更为方便且易于管理，亦可以取代过去 AMD、CommonJS 等规范，成为通用于浏览器端和服务器端的模块化解决方案。但目前浏览器和 Node 在 ES6 模块支持度还不完整，大部分情况需要通过 Babel 转译器进行转译。

Webpack/Browserify + Gulp

随著网页应用程序开发的复杂性提升，现在的网页往往不单只是单纯的网页，而是一个网页应用程序（WebApp）。为了管理复杂的应用程序开发，此时模块化开发方法便显得日益重要，而理想上的模块化开发工具一直是前端工程的很大的议题。Webpack 和 Browserify + Gulp 则是进行 React 应用程序开发常用的开发工具，可以协助进行自动化程序码打包、转译等重复性工作，提升开发效率。本书范例主要会搭配 Webpack 进行开发。

1. Webpack

   Webpack 是一个模块打包工具（module bundler），以下列出 Webpack 的几项主要功能：

   - 将 CSS、图片与其他资源打包
   - 打包之前预处理（Less、CoffeeScript、JSX、ES6 等）的档案
   - 依 entry 文件不同，把 .js 分拆为多个 .js 档案
   - 整合丰富的 Loader 可以使用（Webpack 本身仅能处理 JavaScript 模块，其余档案如：CSS、Image 需要载入不同 Loader 进行处理）

2. Browserify

   如同官网上说明的：`Browserify lets you require('modules') in the browser by bundling up all of your dependencies.`，Browserify 是一个可以让你在浏览器端也能使用像 Node 用的 CommonJS 规范一样，用输出（export）和引用（require）来管理模块。此外，也能让前端使用许多在 NPM 中的模块。

3. Gulp

   `Gulp` 是一个前端任务工具自动化管理工具（Task Runner）。随著前端工程的发展，我们在开发前端应用程序时有许多工作是必须重复进行，例如：打包文件、uglify、将 LESS 转译成一般的 CSS 的档案，转译 ES6 语法等工作。若是使用一般手动的方式，往往会造成效率的低下，所以通过像是 Grunt、Gulp 这类的 Task Runner 不但可以提升效率，也可以更方便管理这些任务。由于 Gulp 是通过 pipeline 方式来处理档案，在使用上比起 Grunt 的方式直观许多，所以这边我们主要讨论的是 Gulp。

ESLint

ESLint 是一个提供 JavaScript 和 JSX 的程序码检查工具，可以确保团队的程序码品质。其支持可插拔的特性，可以根据需求在 `.eslintrc` 设定检查规则。目前主流的检查规则会使用 Airbnb 所发布的 Airbnb React/JSX Style Guide，在使用上需先安装 eslint-config-airbnb 等包。

React Router

React Router 是 React 中主流使用的 Routing 函数库，通过 URL 的变化来管理对应的状态和元件。若开发不刷页的单页式（single page application）的 React 应用程序通常都会需要用到。

Flux/Redux

Flux 是一个实现单项流的应用程序数据架构（architecture），同样是由 Facebook 推出，并和 React 专注于 View 的部份形成互补。而由 Dan Abramov 所开发的 Redux 被 React 开发社群认为是 Flux-like 更优雅的作法，也是目前主流搭配 React 的状态（State）管理工具。让你在开发复杂的应用程序时可以更方便管理你的状态（state）。

ImmutableJS

ImmutableJS，是一个能让开发者建立不可变数据结构的函数库。建立不可变（immutable）数据结构不仅可以让状态可预测性更高，也可以提升程序的性能。

Isomorphic JavaScript

Isomorphic JavaScript 是指前后端（Client/Server）共用相同部分的程序码，让 JavaScript 应用可以同时执行在浏览器端和服务器端，在 React 中可以通过服务器端渲染（server side rendering）静态 HTML 的方式达到 Isomorphic JavaScript 效果，让 SEO 和执行性能更加提升并让前后端共用程序码。而另一个常一起出现的 Universal JavaScript 一般定义更为广泛，系指可以运行在不同环境下的 JavaScript Code，并不局限于浏览器和服务器端。但要留意的是在 Github 和许多技术文章的分享上会把两者定义为同一件事情。

React 测试

Facebook 本身有提供 Test Utilities，但由于不够好用，所以目前主流开发社群比较倾向使用 Airbnb 团队开发的 enzyme，其可以与市面上常见的测试工具（Mocha、Karma、Jest 等）搭配使用。其中 Jest 是 Facebook 所开发的单元测试工具，其主要基于 Jasmine 所建立的测试框架。Jest 除了支持 JSDOM 外，也可以自动模拟 (mock) 通过 `require()` 进来的模块，让开发者可以更专注在目前被测试的模块中。

React Native

React Native和过去的 Apache Cordova 等基于 WebView 的解决方案比较不同，它让开发者可以使用 React 和 JavaScript 开发原生应用程序（Native App），让 `Learn once, write anywhere` 理想变得可能。

GraphQL/Relay

GraphQL 是 Facebook 所开发的数据查询语言（Data Query Language），主要是想解决传统 RESTful API 所遇到的一些问题，并提供前端更有弹性的 API 设计方式。Relay 则是 Facebook 提出搭配 GraphQL 用于 React 的一个定义式数据框架，可以降低 Ajax 的请求数量（类似的框架还有 Netflix 推出的 Falcor）。但由于目前主流的后端 API 仍以传统 RESTful API 设计为主，所以在使用 GraphQL 上通常会需要比较大架构设计的变动。因此本书则是把 GraphQL/Relay 介绍放到附录的部份，让有兴趣的读者可以自行参考体验一下。

总结

以上就是读者在 React 生态系游走时会遇到的各种关卡，也许有些初学者会对于这样庞大的体系所吓到，放弃学习 React 这项革新性技术的机会。不过别担心，接下来笔者将带领读者按图索骥，依序介绍整个 React 生态系的各种技术，一步步带领大家用 React 实作出生活中会用到的应用程序。

延伸阅读

1. Navigating the React.JS Ecosystem
2. petehunt/react-howto
3. React Ecosystem - A summary
4. React Official Site
5. A collection of awesome things regarding React ecosystem
6. Webpack 中文指南
7. AMD 和 CMD 的区别有哪些？
8. jslint to eslint
9. Facebook的Web开发三板斧：React.js、Relay和GraphQL
10. airbnb/javascript

（image via jpsierens）

:door: 任意门

| 回首页 | 上一章：Web 前端工程入门简介 | 下一章：React 开发环境设置与 Webpack 入门教学 |

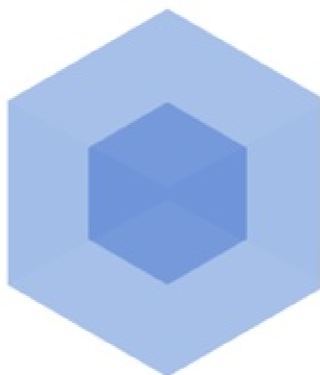
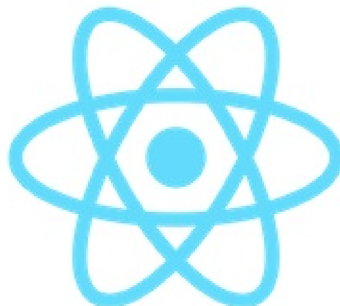
| 纠错、提问或许愿 |

Ch02 React 开发环境设置与 Webpack 入门

:door: 任意门

| 回首页 |

React 开发环境设置与 Webpack 入门教学

前言

俗话说工欲善其事，必先利其器。写程式也是一样，搭建好开发环境后可以让自己在后续开发上更加顺利。因此本章接下来将讨论 React 开发环境的两种主要方式：CDN-based、 webpack（这边我们就先不讨论 TypeScript 的开发方式）。至于 browserify 搭配 Gulp 的方法则会放在补充资料中，让读者阅读完本章后可以开始 React 开发之旅！

JavaScript 模块化

随著网站开发的复杂度提升，许多现代化的网站已不是单纯的网站而已，更像是个富有互动性的网页应用程式（Web App）。为了应付现代化网页应用程式开发的需求，解决一些像是全局变量污染、低维护性等问题，JavaScript 在模块化上也有长足的发展。过去一段时间读者们或许听过像是 `Webpack`、`Browserify`、`module bundlers`、`AMD`、`CommonJS`、`UMD`、`ES6 Module` 等有关 JavaScript 模块化开发的专有名词或工具，在前面一个章节我们也简单介绍了关于 JavaScript 模块化的简单观念和规范介绍。若是读者对于 JavaScript 模块化开发尚不熟悉的话推荐可以参考 这篇文章 和 这篇文章 当作入门。

总的来说，使用模块化开发 JavaScript 应用程式主要有以下三种好处：

1. 提升维护性（Maintainability）

2. 命名空间（Namespacing）
3. 提供可重用性（Reusability）

而在 React 应用程式开发上更推荐使用像是 `Webpack` 这样的 `module bundlers` 来组织我们的应用程式，但对于一般读者来说 `Webpack` 强大而完整的功能相对复杂。为了让读者先熟悉 `React` 核心观念（我们假设读者已经有使用 `JavaScript` 或 `jQuery` 的基本经验），我们将从使用 `CDN` 引入 `<script>` 的方式开始介绍：



使用 CDN-based 的开发方式缺点是较难维护我们的程式码（当引入函式库一多就会有很多 `<script/>` ）且会容易遇到版本相容性问题，不太适合开发大型应用程式，但因为简单易懂，适合教学上使用。

以下是 React 官方首页的范例，以下使用 `React v15.2.1` ：

1. 理解 `React` 是 `Component` 导向的应用程式设计
2. 引入 `react.js` 、 `react-dom.js` （react 0.14 后将 react-dom 从 react 核心分离，更符合 react 跨平台抽象化的定位）以及 `babel-standalone` 版 script（可以想成 `babel` 是翻译机，翻译浏览器看不懂的 `JSX` 或 `ES6+` 语法成为浏览器看的懂得的 `JavaScript` 。为了提升效率，通常我们都会在伺服器端做转译，这点在 production 环境尤为重要）
3. 在 `<body>` 编写 React Component 要插入（mount）指定节点的地方： `<div id="example"></div>`
4. 通过 `babel` 进行语言翻译 `React JSX` 语法， `babel` 会将其转为浏览器看的懂得 `JavaScript` 。其代表意义是： `ReactDOM.render(想要 render`

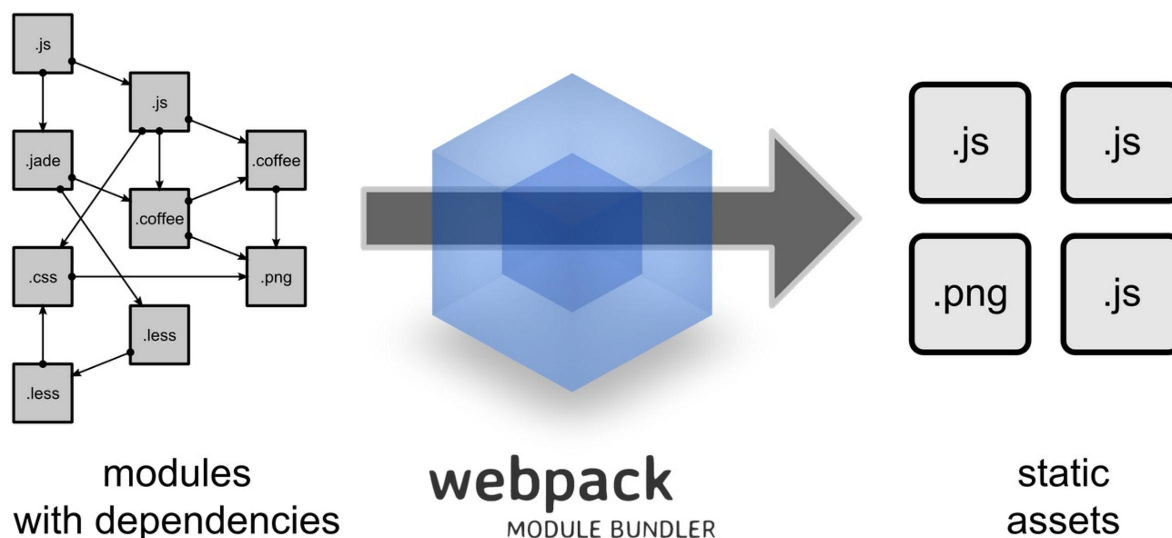
`的 Component 或 HTML 元素, 想要插入的位置)` 。所以我们可以在浏览器上打开我们的 `hello.html` ，就可以看到 `Hello, world!` 。That's it，我们第一个 `React` 应用程式就算完成了！

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <!-- 以下引入 react.js, react-dom.js（react 0.14 后将 react-dom 从 react 核心分离，更符合 react 跨平台抽象化的定位）以及 babel-core browser 版 -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.2.1/react-dom.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-standalone/6.18.1/babel.min.js"></script>
  </head>
  <body>
    <!-- 这边的 id="example" 的 <div> 为 React Component 要插入的地方 -->
    <div id="example"></div>
    <!-- 以下就是包在 babel（通过进行语言翻译）中的 React JSX 语法，babel 会将其转为浏览器看的懂得 JavaScript -->
    <script type="text/babel">
      ReactDOM.render(
                                        ,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

在浏览器浏览最后成果：

Hello, world!

Webpack



上面我们先简单介绍了 CDN-based 的开发方式让大家先对于 React 有个基本印象，但由于 CDN-based 的开发方式有不少缺点。所以接下来的 Webpack 将会是我们接下来范例的主要使用的开发工具。

Webpack 是一个模块打包工具（module bundler），以下列出 Webpack 的几项主要功能：

- 将 CSS、图片与其他资源打包
- 打包之前预处理（Less、CoffeeScript、JSX、ES6 等）档案
- 依 entry 文件不同，把 .js 分拆为多个 .js 档案

- 整合丰富的 Loader 可以使用（Webpack 本身仅能处理 JavaScript 模块，其余档案如：CSS、Image 需要载入不同 Loader 进行处理）

接下来我们一样通过 Hello World 实例来介绍如何用 Webpack 设置 React 开发环境：

1. 依据你的操作系统安装 Node 和 NPM（目前版本的 Node 都会内建 NPM）
2. 通过 NPM 安装 Webpack（可以 global 或 local project 安装，这边我们使用 local）、webpack loader、webpack-dev-server

   Webpack 中的 loader 类似于 browserify 内的 transforms，但 Webpack 在使用上比较多元，除了 JavaScript loader 外也有 CSS Style 和图片的 loader。此外，`webpack-dev-server` 则可以启动开发用 server，方便我们测试

   ```
   // 按指示初始化 NPM 设定档 package.json
   $ npm init
   // --save-dev 是可以让你将安装套件的名称和版本资讯存放到 package.json，方便日后使用
   $ npm install --save-dev babel-core babel-eslint babel-loader babel-preset-es2015 babel-preset-react html-webpack-plugin webpack webpack-dev-server
   ```

3. 在根目录设定 `webpack.config.js`

   事实上，`webpack.config.js` 有点类似于 `gulp` 中的 `gulpfile.js` 功用，主要是设定 `webpack` 的相关设定

   ```
   // 这边使用 HtmlWebpackPlugin，将 bundle 好的 <script> 插入到 body。${__dirname} 为 ES6 语法对应到 __dirname
   const HtmlWebpackPlugin = require('html-webpack-plugin');

   const HTMLWebpackPluginConfig = new HtmlWebpackPlugin({
     template: `${__dirname}/app/index.html`,
     filename: 'index.html',
     inject: 'body',
   });

   module.exports = {
     // 档案起始点从 entry 进入，因为是阵列所以也可以是多个档案
   ```

```
  entry: [
    './app/index.js',
  ],
  // output 是放入产生出来的结果的相关参数
  output: {
    path: `${__dirname}/dist`,
    filename: 'index_bundle.js',
  },
  module: {
      // loaders 则是放想要使用的 loaders，在这边是使用 babel-loader 将所有 .js（这边用到正则）相关文件（排除了 npm 安装的套件位置 node_modules）编译成浏览器可以阅读的 JavaScript。preset 则是使用的 babel 编译规则，这边使用 react、es2015。若是已经单独使用 .babelrc 作为 presets 設定的话，则可以省略 query
    loaders: [
      {
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react'],
        },
      },
    ],
  },
  // devServer 则是 webpack-dev-server 设定
  devServer: {
    inline: true,
    port: 8008,
  },
  // plugins 放置所使用的外挂
  plugins: [HTMLWebpackPluginConfig],
};
```

4. 在根目录设定 `.babelrc`

```
{
  "presets": [
    "es2015",
    "react",
  ],
  "plugins": []
}
```

5. 安装 react 和 react-dom

```
$ npm install --save react react-dom
```

6. 编写 Component（记得把 `index.html` 以及 `index.js` 放到 `app` 文件夹底下喔！）

   `index.html`

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>React Setup</title>
    <link rel="stylesheet" type="text/css" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css">
</head>
<body>
    <!-- 想要插入 React Component 的位置 -->
    <div id="app"></div>
</body>
</html>
```

   `index.js`

```
import React from 'react';
import ReactDOM from 'react-dom';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
    };
  }
  render() {
    return (
      <div>
        <h1>Hello, World!</h1>
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('app'));
```

7. 在终端机使用 `webpack` 进行成果展示，webpack 相关指令：

   - webpack：会在开发模式下开始一次性的建置
   - webpack -p：会建置 production 的程式码
   - webpack --watch：会监听程式码的修改，当储存时有异动时会更新档案
   - webpack -d：加入 source maps 档案
   - webpack --progress --colors：加上处理进度与颜色

   如果不想每次都打一长串的指令码的话可以使用 `package.json` 中的 `scripts` 设定

   ```
   "scripts": {
   "dev": "webpack-dev-server --devtool eval --progress --colors --content-base build"
   }
   ```

   然后在终端机执行：

```
$ npm run dev
```

当我们此时我们可以打开浏览器输入 `http://localhost:8008` ，就可以看到 `Hello, world!` 了！

总结

以上就是 React 开发环境设置与 Webpack 入门教学，看到这边的读者不妨自己动手设定开发环境，体验一下 React 开发环境的感觉，毕竟若是只有阅读文字的话很容易就会忘记喔！若你不想在环境设定上花太多时间的话，不妨参考 Facebook 开发社区推出的 create-react-app，可以快速上手，使用 Webpack、Babel、ESLint 开发 React 应用程式。接下来的章节我们将持续延伸 React/JSX/Component 的介绍。

延伸阅读

1. JavaScript 模块化七日谈
2. 前端模块化开发那点历史
3. Webpack 中文指南
4. WEBPACK DEV SERVER

(image via srinisoundar、sitepoint、keyholesoftware、survivejs)

:door: 任意门

| 回首页 | 上一章：React 生态系（Ecosystem）入门简介 | 下一章：ReactJS 与 Component 设计入门介绍 |

| 纠错、提问或许愿 |

Ch03 React/JSX/Component 简介

:door: 随意门

| 回首页 |

ReactJS 与 Component 设计入门介绍

前言

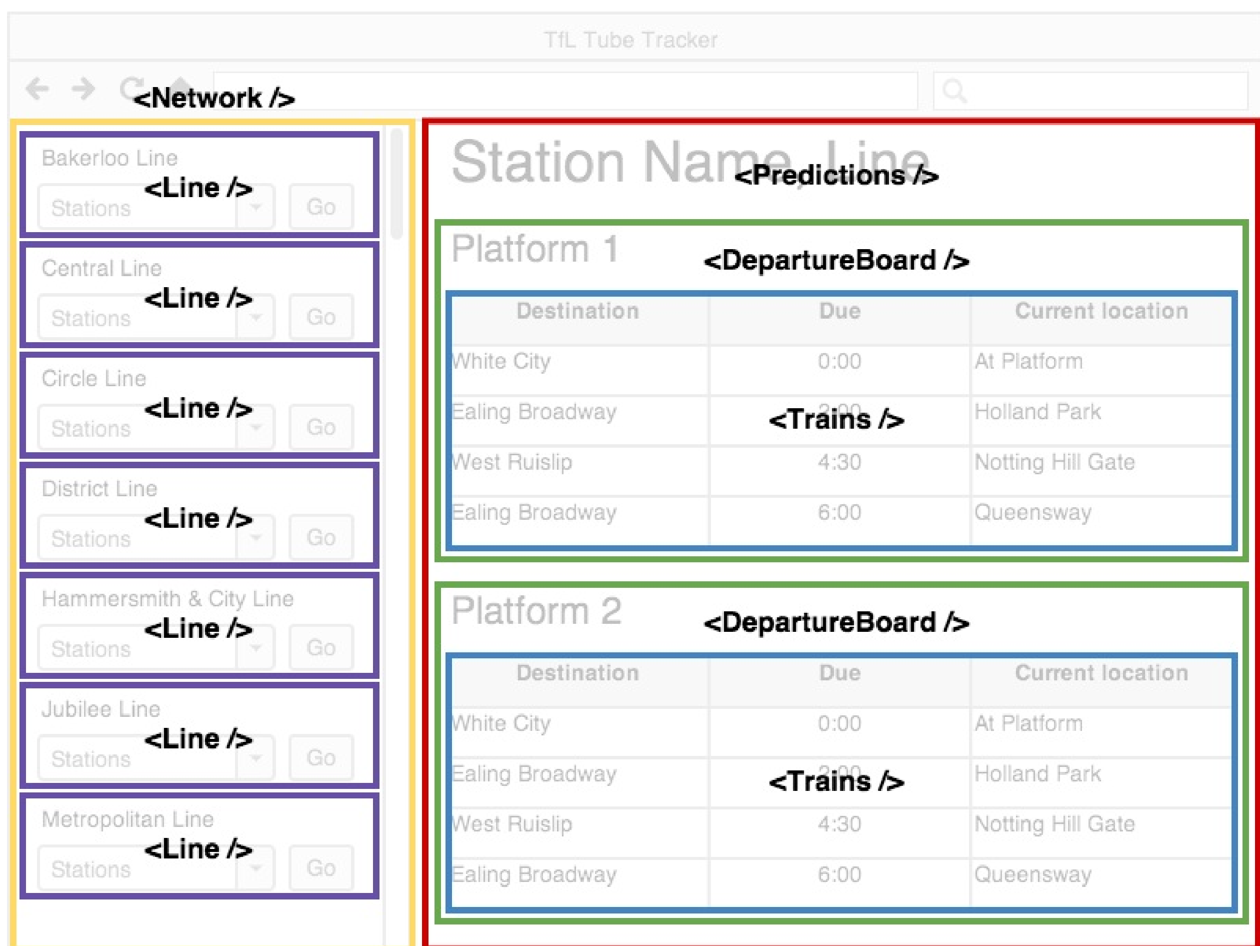
在上一个章节中我们快速学习了 React 开发环境建置和 Webpack 入门。接下来我们将更进一步了解 React 和 Component 设计时需注意的几个重要特性。

ReactJS 特性简介

React 原本是 Facebook 自己内部使用的开发工具，但却是一个目标远大的一个项目： `Learn once, write anywhere` 。自从 2013 年开源后周边的生态系更是蓬勃发展。ReactJS 的出现让前端开发有许多革新性的思维出现，其中有几个重要特性值得我们去探讨：

1. 基于组件（Component）化思考
2. 用 JSX 进行声明式（Declarative）UI 设计
3. 使用 Virtual DOM
4. Component PropType 错误校对机制
5. Component 就像个状态机（State Machine），而且也有生命周期（Life Cycle）
6. 一律重绘（Always Redraw）和单向数据流（Unidirectional Data Flow）
7. 在 JavaScript 里写 CSS：Inline Style

基于组件（Component）化思考



在 React 的世界中最基本的单元为组件（Component），每个组件也可以包含一个以上的子组件，并依照需求组装成一个组合式的（Composable）组件，因此具有封装（encapsulation）、关注点分离 (Separation of Concerns)、复用 (Reuse) 、组合 (Compose) 等特性。

`<TodoApp>` 组件可以包含 `<TodoHeader />` 、 `<TodoList />` 子组件

```
<div>
    <TodoHeader />
    <TodoList />
</div>
```

`<TodoList />` 组件内部长相：

```
    <div>
        <ul>
            <li>写程式码</li>
            <li>哄妹子</li>
            <li>买书</li>
        </ul>
    </div>
```

组件化一直是网页前端开发的万金油，许多开发者最希望的就是可以最大化重复使用（reuse）过去所写的程式码，不要重复造轮子（DRY）。在 React 中组件是一切的基础，让开发应用程式就好像在堆积木一样。然而对于过去习惯模版式（template）开发的前端工程师来说，短时间要转换成组件化思考模式并不容易，尤其过去我们往往习惯于将 HTML、CSS 和 JavaScript 分离，现在却要把它们都封装在一起。

一个比较好的方式就是训练自己看到不同的网页或应用程式时，强迫自己将看到的页面切成一个个组件。相信过了一段时间后，天眼开了，就比较容易习惯组件化的思考方式。

以下是一般 React Component 撰写的主要两种方式：

1. 使用 ES6 的 Class（可以进行比较复杂的操作和组件生命周期的控制，相对于 stateless components 耗费资源）

```
// 注意组件开头第一个字母都要大写
class MyComponent extends React.Component {
    // render 是 Class based 组件唯一必须的方法（method）
    render() {
        return (
            <div>Hello, World!</div>
        );
    }
}

// 将 <MyComponent /> 组件插入 id 为 app 的 DOM 元素中
ReactDOM.render(<MyComponent/>, document.getElementById('app'));
```

<<<<<<< HEAD

1. 使用 Funtional Component 写法（单纯地 render UI 的 stateless components，没有内部状态、没有实作物件和 ref，没有生命周期函数。若非需要控制生命周期的话建议多使用 stateless components 获得比较好的性能）

   ```javascript

# // 使用 arror function 来设计 Funtional Component 让 UI 设计更单纯（f(D) => UI），减少副作用（side effect）

2. 使用 Functional Component 寫法（單純地 render UI 的 stateless components，沒有內部狀態、沒有實作物件和 ref，沒有生命週期函數。若非需要控制生命週期的話建議多使用 stateless components 獲得比較好的效能）

```
 // 使用 arrow function 來設計 Functional Component 讓 UI 設計更單純（f(D) => UI），減少副作用（side effect）
>>>>>>> kdchang/master
 const MyComponent = () => (
     Hello, World!
 );

 // 将 <MyComponent /> 组件插入 id 为 app 的 DOM 元素中
 ReactDOM.render(<MyComponent/>, document.getElementById('app'));
```

## 用 JSX 进行声明式（Declarative）UI 设计

React 在设计上的思路认为使用 Component 比起模版（Template）和显示逻辑（Display Logic）更能实现关注点分离的概念，而搭配 JSX 可以实现声明式 Declarative（注重 what to），而非命令式 Imperative（注重 how to）的程式撰写方式。

像下述的声明式（Declarative）UI 设计就比单纯用（Template）式的方式更易懂：
```

```
// 使用声明式（Declarative）UI 设计很容易可以看出这个组件的功能
<MailForm />
```

```
// <MailForm /> 内部长相
<form>
    <input type="text" name="email">
    <button type="submit"></button>
</form>
```

由于 JSX 在 React 组件撰写上扮演很重要的角色，因此在下一个章节我们也将更深入讲解 JSX 使用细节。

使用 Virtual DOM

在传统 Web 中一般是使用 jQuery 进行 DOM 的直接操作。然而更改 DOM 往往是 Web 性能的瓶颈，因此在 React 世界设计有 Virtual DOM 的机制，让 App 和 DOM 之间用 Virtual DOM 进行沟通。当更改 DOM 时，会通过 React 自身的 diff 演算法去计算出最小更新，进而去最小化更新真实的 DOM。

Component PropType 错误校对机制

在 React 设计时除了提供 props 预设值设定（Default Prop Values）外，也提供了 Prop 的验证（Validation）机制，让整个 Component 设计更加稳健：

```
//  注意组件开头第一个字母都要大写
class MyComponent extends React.Component {
    // render 是 Class based 组件唯一必须的方法（method）
    render() {
        return (
            <div>Hello, World!</div>
        );
    }
}

// PropTypes 验证，若传入的 props type 不符合将会显示错误
MyComponent.propTypes = {
  todo: React.PropTypes.object,
  name: React.PropTypes.string,
}

// Prop 预设值，若对应 props 没传入值将会使用 default 值
MyComponent.defaultProps = {
 todo: {},
 name: '',
}
```

关于更多的 Validation 用法可以参考官方网站 的说明。

Component 就像个状态机（State Machine），而且也有生命周期（Life Cycle）

Component 就像个状态机（State Machine），根据不同的 state（通过 `setState()` 修改）和 props（由父元素传入），Component 会出现对应的显示结果。而人有生老病死，组件也有生命周期。通过操作生命周期处理函数，可以在对应的时间点进行 Component 需要的处理，关于更详细的组件生命周期介绍我们会再下一个章节进行更一步说明。

一律重绘（Always Redraw）和单向数据流（Unidirectional Data Flow）

在 React 世界中，props 和 state 是影响 React Component 长相的重要要素。其中 props 都是由父元素所传进来，不能更改，若要更改 props 则必须由父元素进行更改。而 state 则是根据使用者互动而产生的不同状态，主要是通过 setState() 方法进行修改。当 React 发现 props 或是 state 更新时，就会重绘整个 UI。当然你也可以使用 forceUpdate() 去强迫重绘 Component。而 React 通过整合 Flux 或 Flux-like（例如：Redux）可以更具体实现单向数据流（Unidirectional Data Flow），让数据流的管理更为清晰。

在 JavaScript 里写 CSS：Inline Style

在 React Component 中 CSS 使用 Inline Style 写法，全都封装在 JavaScript 当中：

```
const divStyle = {
  color: 'red',
  backgroundImage: 'url(' + imgUrl + ')',
};

ReactDOM.render(<div style={divStyle}>Hello World!</div>, docume
nt.getElementById('app'));
```

总结

以上介绍了 ReactJS 的几个重要特性：

1. 基于组件（Component）化思考
2. 用 JSX 进行声明式（Declarative）UI 设计
3. 使用 Virtual DOM
4. Component PropType 错误校对机制
5. Component 就像个状态机（State Machine），而且也有生命周期（Life Cycle）
6. 一律重绘（Always Redraw）和单向数据流（Unidirectional Data Flow）
7. 在 JavaScript 里写 CSS：Inline Style

接下来我们将进一步探讨 React 里 JSX 的使用方式。

延伸阅读

1. React 入门实例教程
2. React Demystified
3. Top-Level API
4. ES6 Classes Component

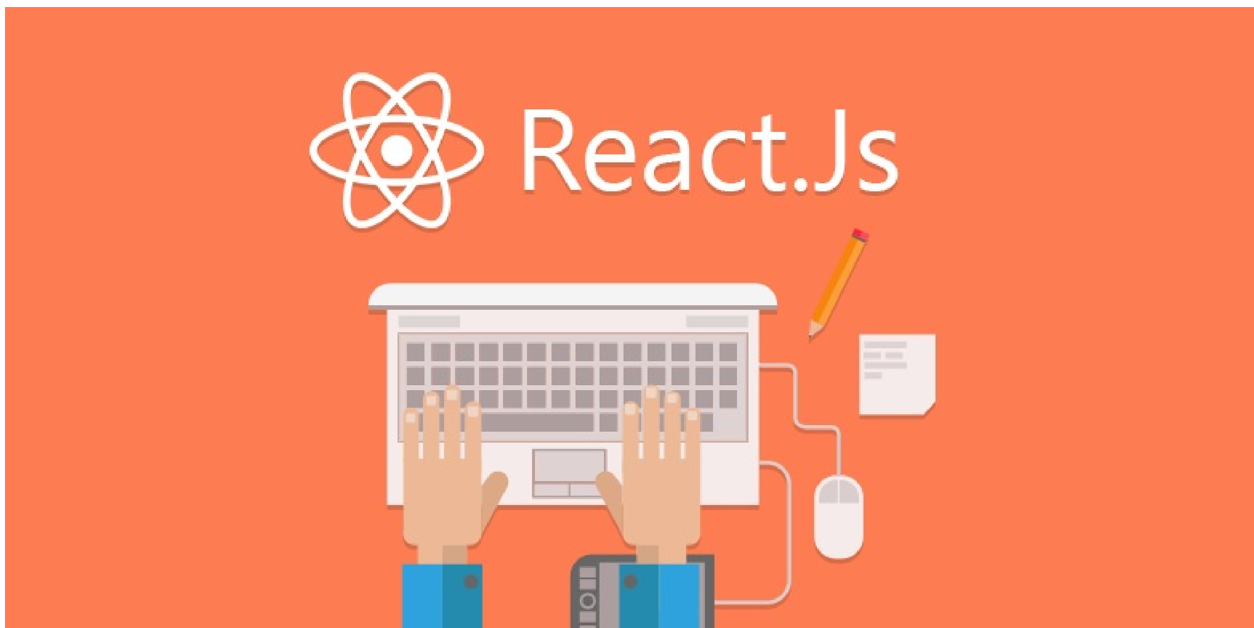
（image via maketea）

:door: 任意门

| 回首页 | 上一章：React 开发环境设置与 Webpack 入门教学 | 下一章：JSX 简明入门教学指南 |

| 纠错、提问或许愿 |

JSX 简明入门教学指南



前言

根据 React 官方定义，React 是一个构建使用者介面的 JavaScritp Library。以 MVC 模式来说，ReactJS 主要是负责 View 的部份。过去一段时间，我们被灌输了许多前端分离的观念，在前端三兄弟中（或三姊妹、三剑客）：HTML 掌管内容结构、CSS 负责外观样式，JavaScript 主管逻辑互动，千万不要混在一块。然而，在 React 世界里，所有事物都是 以 Component 为基础，将同一个 Compoent 相关的程序和资源都放在一起，而在编写 React Component 时我们通常会使用 JSX 的方式来提升程序编写效率。事实上，JSX 并非一种全新的语言，而是一种语法糖（Syntatic Sugar），一种语法类似 XML 的 ECMAScript 语法扩充。在 JSX 中 HTML 和组建这些元素标签的代码有紧密的关系。因此你可能要熟悉一下以 Component 为单位的思考方式（本文主要使用 ES6 语法）。

此外，React 和 JSX 的思维在于善用 JavaScript 的强大能力，放弃蹩脚的模版语言，这和 Angular 强化 HTML 的理念也有所不同。当然 JSX 并非强制使用，你也可以选择不用，因为最终 JSX 的内容会转化成 JavaScript（浏览器只看的懂 JavaScript）。不过等你阅读完接下来的内容，你或许会开始发现 JSX 的好，认真考虑使用 JSX 的语法。

一、使用 JSX 的好处

1. 提供更加语意化且易懂的标签

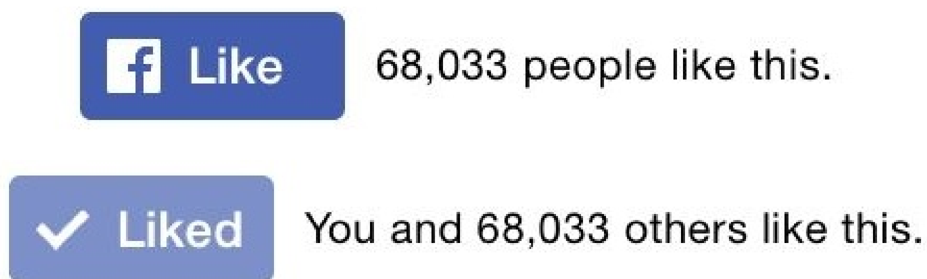
由于 JSX 类似 XML 的语法，让一些非开发人员也更容易看懂，且能精确定义包含属性的树状结构。一般来说我们想做一个反馈表单，使用 HTML 写法通常会长这样：

```
<form class="messageBox">
  <textarea></textarea>
  <button type="submit"></button>
</form>
```

使用 JSX，就像 XML 语法结构一样可以自行定义标签且有开始和关闭，容易理解：

```
<MessageBox />
```

React 思路认为使用 Component 比起模版（Template）和显示逻辑（Display Logic）更能实现关注点分离的概念，而搭配 JSX 可以实现声明式 `Declarative` （注重 what to），而非命令式 `Imperative` （注重 how to）的程序编写方式：



以 Facebook 上面点赞功能来说，若是命令式 `Imperative` 写法大约会是长这样：

```
if(userLikes()) {
  if(!hasBlueLike()) {
    removeGrayLike();
    addBlueLike();
  }
} else {
  if(hasBlueLike()) {
    removeBlueLike();
    addGrayLike();
  }
}
```

若是声明式 `Declarative` 则是会长这样：

```
if(this.state.liked) {
  return (<BlueLike />);
} else {
  return (<GrayLike />);
}
```

看完上述说明是不是感觉 `React` 结合 `JSX` 的写法更易读易懂？事实上，当 Component 组成越来越复杂时，若使用 JSX 将可以让整个结构更加直观，可读性较高。

2. 更加简洁

虽然最终 JSX 会转换成 JavaScript，但使用 JSX 可以让程序看起来更加简洁，以下为使用 JSX 和不使用 JSX 的范例：

```
<a href="https://facebook.github.io/react/">Hello!</a>
```

不使用 JSX（记得我们说过 JSX 是选用的）：

```
// React.createElement(元件/HTML标签, 元件属性，以对象表示, 子元件)
React.createElement('a', {href: 'https://facebook.github.io/reac
t/'}, 'Hello!')
```

3. 结合原生 JavaScript 语法

JSX 并非一种全新的语言，而是一种语法糖（Syntatic Sugar），一种语法类似 XML 的 ECMAScript 语法扩充，所以并没有改变 JavaScript 语意。通过结合 JavaScript ，可以释放 JavaScript 语言本身能力。下面例子就是运用 `map` 方法，轻易把 `result` 值迭代出来，产生无序清单（ul）的内容，不用再使用蹩脚的模版语言（这边有个小地方要留意的是每个 `<li>` 元素记得加上独特的 key 这边用 map function 迭代出的 index，不然会出现问题）：

```
// const 为常数
const lists = ['JavaScript', 'Java', 'Node', 'Python'];

class HelloMessage extends React.Component {
  render() {
    return (
    <ul>
      {lists.map((result, index) => {
        return (<li key={index}>{result}</li>);
      })}
    </ul>);
  }
}
```

二、JSX 用法摘要

1. 环境设定与使用方式

初步了解为何要使用 JSX 后，我们来聊聊 JSX 的用法。一般而言 JSX 通常有两种使用方式：

1. 使用 browserify 或 webpack 等 CommonJS bundler 并整合 babel 预处理
2. 于浏览器端做解析

在这边简单起见，我们先使用第二种方式，先让大家专注熟悉 JSX 语法使用，等到后面章节再教大家使用 bundler 的方式去做解析（可以试著把下面的原始码贴到 JSbin 的 HTML 看结果）：

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <!-- 请先于 index.html 中引入 react.js, react-dom.js 和 babel-core 的 browser.min.js -->
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.0.1/react.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/15.0.1/react-dom.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      // 代码写在这边！
      ReactDOM.render(
                                ,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

一般加载 JSX 方式有：

- 内嵌

```
<script type="text/babel">
  ReactDOM.render(
                          ,
    document.getElementById('example')
  );
</script>
```

- 从外部引入

```
<script type="text/jsx" src="main.jsx"></script>
```

2. 标签用法

JSX 标签非常类似 XML ，可以直接书写。一般 Component 命名首字大写，HTML Tags 小写。以下是一个建立 Component 的 class：

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        <p>Hello React!</p>
        <MessageList />
      </div>
    );
  }
}
```

3. 转换成 JavaScript

JSX 最终会转换成浏览器可以读取的 JavaScript，以下为其规则：

```
React.createElement(
  string/ReactClass, // 表示 HTML 元素或是 React Component
  [object props], // 属性值，用对象表示
  [children] // 接下来参数皆为元素子元素
)
```

解析前（特别注意在 JSX 中使用 JavaScript 表达式时使用 `{}` 括起，如下方范例的 `text` ，里面对应的是变数。若需希望放置一般文字，请加上 `''` ）：

```
var text = 'Hello React';
<h1>{text}</h1>
<h1>{'text'}</h1>
```

解析完后：

```
var text = 'Hello React';
React.createElement("h1", null, "Hello React!");
```

另外要特别要注意的是由于 JSX 最终会转成 JavaScript 且每一个 JSX 节点都对应到一个 JavaScript 函数，所以在 Component 的 `render` 方法中只能回传一个根节点（Root Nodes）。例如：若有多个 `<div>` 要 `render` 请在外面包一个 Component 或 `<div>` 、 `<span>` 元素。

4. 注解

由于 JSX 最终会编译成 JavaScript，注解也一样使用 `//` 和 `/**/` 当做注解方式：

```
// 单行注解

/*
  多行注解
*/

var content = (
  <List>
      {/* 若是在子元件注解要加 {}  */}
      <Item
        /* 多行
           注解
           喔 */
        name={window.isLoggedIn ? window.name : ''} // 单行注解
      />
  </List>
);
```

5. 属性

在 HTML 中，我们可以通过标签上的属性来改变标签外观样式，在 JSX 中也可以，但要注意 `class` 和 `for` 由于为 JavaScript 保留关键字用法，因此在 JSX 中使用 `className` 和 `htmlFor` 替代。

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div className="message">
        <p>Hello React!</p>
      </div>
    );
  }
}
```

Boolean 属性

在 JSX 中预设只有属性名称但没设值为 `true` ，例如以下第一个 input 标签 `disabled` 虽然没设值，但结果和下面的 input 为相同：

```
<input type="button" disabled />;
<input type="button" disabled={true} />;
```

反之，若是没有属性，则预设预设为 `false` ：

```
<input type="button" />;
<input type="button" disabled={false} />;
```

6. 扩展属性

在 ES6 中使用 `...` 是迭代对象的意思，可以把所有对象对应的值迭代出来设定属性，但要注意后面设定的属性会盖掉前面相同属性：

```
var props = {
  style: "width:20px",
  className: "main",
  value: "yo",
}

<HelloMessage  {...props} value="yo" />

// 等于以下
React.createElement("h1", React._spread({}, props, {value: "yo"}
), "Hello React!");
```

7. 自定义属性

若是希望使用自定义属性，可以使用 `data-` ：

```
<HelloMessage data-attr="xd" />
```

8. 显示 HTML

通常为了避免信息安全问题，我们会过滤掉 HTML，若需要显示的话可以使用：

```
<div>{{_html: '<h1>Hello World!!</h1>'}}</div>
```

9. 样式使用

在 JSX 中使用外观样式方法如下，第一个 `{}` 是 JSX 语法，第二个为 JavaScript 对象。与一般属性值用 `-` 分隔不同，为驼峰式命名写法：

```
<HelloMessage style={{ color: '#FFFFFF', fontSize: '30px'}} />
```

10. 事件处理

事件处理为前端开发的重头戏，在 JSX 中通过 inline 事件的绑定来监听并处理事件（注意也是驼峰式写法），更多事件处理方法请参考官网

```
<HelloMessage onClick={this.onBtn} />
```

总结

以上就是 JSX 简明入门教学，希望通过以上介绍，让读者了解在 React 中为何要使用 JSX，以及 JSX 基本概念和用法。最后为大家复习一下：在 React 世界里，所有事物都是以 Component 为基础，通常会将同一个 Compoent 相关的程序和资源都放在一起，而在编写 React Component 时我们常会使用 JSX 的方式来提升程序编写效率。JSX 是一种语法类似 XML 的 ECMAScript 语法扩充，可以善用 JavaScript 的强大能力，放弃蹩脚的模版语言。当然 JSX 并非强制使用，你也可以选择不用，因为最终 JSX 的内容会转化成 JavaScript。当相信阅读完上述的内容后，你会开始认真考虑使用 JSX 的语法。

延伸阅读

1. Imperative programming or declarative programming
2. JSX in Depth
3. 从零开始学 React（ReactJS 101）

（image via adweek, codecondo）

:door: 任意门

| 回首页 | 上一章：ReactJS 与 Component 设计入门介绍 | 下一章：Props、State、Refs 与表单处理 |

| 纠错、提问或许愿 |

Ch04 Props/State 基础与 Component 生命周期

1. Props、State、Refs 与表单处理
2. React Component 规格与生命周期（Life Cycle）

:door: 随意门

| 回首页 |

Props、State、Refs 与表单处理

前言

在前面的章节中我们已经对于 React 和 JSX 有初步的认识，我们也了解到 React Component 事实上可以视为显示 UI 的一个状态机（state machine），而这个状态机根据不同的 state（通过 `setState()` 修改）和 props（由父元素传入），Component 会出现对应的显示结果。本章将使用 React 官网首页上的范例（使用 ES6+）来更进一步说明 Props 和 State 特性及在 React 如何进行事件和表单处理。

Props

首先我们使用 React 官网上的 A Simple Component 来说明 props 的使用方式。由于传入组件的 name 属性为 Mark，故以下代码将会在浏览器显示 Hello, Mark。针对传入的 props 我们也有验证和预设值的设计，让我们撰写的组件可以更加稳定健壮（robust）。

HTML Markup：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>A Component Using External Plugins</title>
</head>
<body>
<!-- 这边方便使用 CDN 方式引入 react 、 react-dom 进行讲解，实务上和实
战教学部分我们会使用 webpack -->
<script src="https://fb.me/react-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
  <div id="app"></div>
    <script src="./app.js"></script>
</body>
</html>
```

app.js，使用 ES6 Class Component 写法：

```
class HelloMessage extends React.Component {
    // 若是需要綁定 this.方法或是需要在 constructor 使用 props，定義 state，就需要 constructor。若是在其他方法（如 render）使用 this.props 則不用一定要定義 constructor
    constructor(props) {
        // 对于 OOP 面向对象程序设计熟悉的读者应该对于 constructor 建构子的使用不陌生，事实上它是 ES6 的语法糖，骨子里还是 prototype based 面向对象程序语言。通过 extends 可以继承 React.Component 父类别。super 方法可以调用继承父类别的建构子
        super(props);
        this.state = {}
    }
    // render 是唯一必須的方法，但如果是單純 render UI 建議使用 Functional Component 寫法，效能較佳且較簡潔
    render() {
        return (
            <div>Hello {this.props.name}</div>
        )
    }
}

// PropTypes 验证，若传入的 props type 不是 string 将会显示错误
HelloMessage.propTypes = {
  name: React.PropTypes.string,
}

// Prop 预设值，若对应 props 没传入值将会使用 default 值 Zuck
HelloMessage.defaultProps = {
 name: 'Zuck',
}

ReactDOM.render(<HelloMessage name="Mark" />, document.getElementById('app'));
```

关于 React ES6 class constructor super() 解释可以参考 React ES6 class constructor super() 。

使用 Functional Component 写法：

```
// Functional Component 可以视为 f(d) => UI，根据传进去的 props 绘出
对应的 UI。注意这边 props 是传入函式的参数，因此取用 props 不用加 this
const HelloMessage = (props) => (
    <div>Hello {props.name}</div>
);

// PropTypes 验证，若传入的 props type 不是 string 将会显示错误
HelloMessage.propTypes = {
  name: React.PropTypes.string,
}

// Prop 预设值，若对应 props 没传入值将会使用 default 值 Zuck。用法等于
 ES5 的 getDefaultProps
HelloMessage.defaultProps = {
 name: 'Zuck',
}

ReactDOM.render(<HelloMessage name="Mark" />, document.getElemen
tById('app'));
```

在 jsbin 上的范例：

A Component Using External Plugins on jsbin.com

State

接下来我们将使用 A Stateful Component 这个范例来讲解 State 的用法。在 React Component 可以自己管理自己的内部 state，并用 `this.state` 来存取 state。当 `setState()` 方法更新了 state 后将重新调用 `render()` 方法，重新绘制 component 内容。以下范例是一个每 1000 毫秒（等于1秒）就会加一的累加器。由于这个范例是 Stateful Component 因此仅使用 ES6 Class Component，而不使用 Functional Component。

HTML Markup：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>A Component Using External Plugins</title>
</head>
<body>
<script src="https://fb.me/react-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
  <div id="app"></div>
    <script src="./app.js"></script>
</body>
</html>
```

app.js：

```
class Timer extends React.Component {
    constructor(props) {
        super(props);
        // 与 ES5 React.createClass({}) 不同的是 component 内自定义
的方法需要自行绑定 this context，或是使用 arrow function
        this.tick = this.tick.bind(this);
        // 初始 state，等于 ES5 中的 getInitialState
        this.state = {
            secondsElapsed: 0,
        }
    }
    // 累加器方法，每一秒被调用后就会使用 setState() 更新内部 state，让
Component 重新 render
    tick() {
        this.setState({secondsElapsed: this.state.secondsElapsed
 + 1});
    }
    // componentDidMount 为 component 生命周期中阶段 component 已插
入节点的阶段，通常一些非同步操作都会放置在这个阶段。这便是每1秒钟会去调用 ti
ck 方法
    componentDidMount() {
        this.interval = setInterval(this.tick, 1000);
```

```
    }
    // componentWillUnmount 为 component 生命周期中 component 即将
移出插入的节点的阶段。这边移除了 setInterval 效力
    componentWillUnmount() {
        clearInterval(this.interval);
    }
    // render 为 class Component 中唯一需要定义的方法，其回传 compone
nt 欲显示的内容
    render() {
        return (
          <div>Seconds Elapsed: {this.state.secondsElapsed}</div>

        );
    }
}

ReactDOM.render(<Timer />, document.getElementById('app'));
```

事件处理（Event Handle）

在前面的内容我们已经学会如何使用 props 和 state，接下来我们要更进一步学习在 React 内如何进行事件处理。下列将使用 React 官网的 An Application 当做例子，实践出一个简单的 TodoApp。

HTML Markup：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>A Component Using External Plugins</title>
</head>
<body>
<script src="https://fb.me/react-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
  <div id="app"></div>
    <script src="./app.js"></script>
</body>
</html>
```

app.js：

```
// TodoApp 组件中包含了显示 Todo 的 TodoList 组件，Todo 的内容通过 pro
ps 传入 TodoList 中。由于 TodoList 仅单纯 Render UI 不涉及内部 state
操作是 stateless component，所以使用 Functional Component 写法。需要
特别注意的是这边我们用 map function 来迭代 Todos，需要留意的是每个迭代的
元素必须要有 unique key 不然会发生错误（可以用自定义 id，或是使用 map fu
nction 的第二个参数 index）
const TodoList = (props) => (
    <ul>
        {
            props.items.map((item) => (
                <li key={item.id}>{item.text}</li>
            ))
        }
    </ul>
)

// 整个 App 的主要组件，这边比较重要的是事件处理的部份，内部有
class TodoApp extends React.Component {
    constructor(props) {
        super(props);
        this.onChange = this.onChange.bind(this);
        this.handleSubmit = this.handleSubmit.bind(this);
```

```
        this.state = {
            items: [],
            text: '',
        }
    }
    onChange(e) {
        this.setState({text: e.target.value});
    }
    handleSubmit(e) {
        e.preventDefault();
        const nextItems = this.state.items.concat([{text: this.s
tate.text, id: Date.now()}]);
        const nextText = '';
        this.setState({items: nextItems, text: nextText});
    }
    render() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <input onChange={this.onChange} value={this.state.text
} />
          <button>{'Add #' + (this.state.items.length + 1)}</but
ton>
        </form>
      </div>
    );
    }
}

ReactDOM.render(<TodoApp />, document.getElementById('app'));
```

以上介绍了 React 事件处理的部份，除了 `onChange` 和 `onSubmit` 外，React 也封装了常用的事件处理，如 `onClick` 等。若想更进一步了解有哪些可以使用的事件处理方法可以参考 官网的 Event System。

Refs 与表单处理

上面介绍了 props（传入后就不能修改）、state（随著使用者互动而改变）和事件处理机制后，我们将接续介绍如何在 React 中进行表单处理。同样我们使用 React 官网范例 A Component Using External Plugins 进行介绍。由于 React 可以容易整合外部的 libraries（例如：jQuery），本范例将使用 `remarkable` 结合 `ref` 属性取出 DOM Value 值（另外比较常用的作法是使用 `onChange` 事件处理方式处理表单内容），让使用者可以使用 Markdown 语法的所见即所得编辑器（editor）。

HTML Markup (除了引入 `react` 、 `react-dom` 还要用 `CDN` 方式引入 `remarkable` 这个 `Markdown` 语法 parser 套件，记得如果没有使用 Webpack 或是 browserify + babelify 等工具需要引入 `babel-standalone` 浏览器解析 ES6 读法并引入 script 加上 type="text/babel")：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <title>A Component Using External Plugins</title>
</head>
<body>
<script src="https://fb.me/react-15.1.0.js"></script>
<script src="https://fb.me/react-dom-15.1.0.js"></script>
<script src="https://cdn.jsdelivr.net/remarkable/1.6.2/remarkabl
e.min.js"></script>
  <div id="app"></div>
    <script type="text/babel" src="./app.js"></script>
</body>
</html>
```

app.js：

```
class MarkdownEditor extends React.Component {
    constructor(props) {
        super(props);
        this.handleChange = this.handleChange.bind(this);
        this.rawMarkup = this.rawMarkup.bind(this);
        this.state = {
```

```
                value: 'Type some *markdown* here!',
            }
        }
        handleChange() {
            this.setState({value: this.refs.textarea.value});
        }
        // 将使用者输入的 Markdown 语法 parse 成 HTML 放入 DOM 中，React
    通常使用 virtual DOM 作为和 DOM 沟通的中介，不建议直接由操作 DOM。故使用
    时的属性为 dangerouslySetInnerHTML
        rawMarkup() {
            const md = new Remarkable();
            return { __html: md.render(this.state.value) };
        }
        render() {
            return (
              <div className="MarkdownEditor">
                <h3>Input</h3>
                <textarea
                  onChange={this.handleChange}
                  ref="textarea"
                  defaultValue={this.state.value} />
                <h3>Output</h3>
                <div
                  className="content"
                  dangerouslySetInnerHTML={this.rawMarkup()}
                />
              </div>
            );
        }
}

ReactDOM.render(<MarkdownEditor />, document.getElementById('app
'));
```

总结

以上通过几个 React 官网首页上的范例介绍了 Props 和 State 特性及在 React 如何进行事件和表单处理这些 React 中核心的问题，若还不熟悉的读者建议重新亲自动手照著范例中的代码敲过一遍，也可以使用像 jsbin 这样所见即所得的工具来练习，更能熟悉相关语法和 API 喔！接下来我们将探讨 Component 的生命周期。

延伸阅读

1. React 官方网站
2. Top-Level API
3. Javascript：this用法整理

:door: 任意门

| 回首页 | 上一章：JSX 简明入门教学指南 | 下一章：React Component 规格与生命周期（Life Cycle） |

| 纠错、提问或许愿 |

React Component 规格与生命周期（Life Cycle）

前言

经过前面的努力相信目前读者对于用 React 开发一些简单的组件（Component）已经有一定程度的掌握了，现在我们将更细部探讨 React Component 的规格和其生命周期。

React Component 规格

若读者还有印象的话，我们前面介绍 React 特性时有描述 React 的主要编写方式有两种：一种是使用 ES6 Class，另外一种是 Stateless Components，使用 Functional Component 的写法，单纯渲染 UI。这边再帮大家复习一下上一个章节的简单范例：

1. 使用 ES6 的 Class（可以进行比较复杂的操作和组件生命周期的控制，相对于 stateless components 耗费资源）

```
//  注意组件开头第一个字母都要大写
class MyComponent extends React.Component {
 // render 是 Class based 组件唯一必须的方法（method）
 render() {
   return (
     <div>Hello, {this.props.name}</div>
   );
 }
}

// PropTypes 验证，若传入的 props type 不符合将会显示错误
MyComponent.propTypes = {
 name: React.PropTypes.string,
}

// Prop 预设值，若对应 props 没传入值将会使用 default 值，为每个实例
化 Component 共用的值
MyComponent.defaultProps = {
 name: '',
}

// 将 <MyComponent /> 组件插入 id 为 app 的 DOM 元素中
ReactDOM.render(<MyComponent name="Mark"/>, document.getElme
ntById('app'));
```

2. 使用 Functional Component 写法（单纯地 render UI 的 stateless components，没有内部状态、没有实际对象和 ref，没有生命周期函数。若非需要控制生命周期的话建议多使用 stateless components 获得比较好的效能）

```
// 使用 arrow function 来设计 Functional Component 让 UI 设计更单纯（f(D) => UI），减少副作用（side effect）
const MyComponent = (props) => (
 <div>Hello, {props.name}</div>
);

// PropTypes 验证，若传入的 props type 不符合将会显示错误
MyComponent.propTypes = {
 name: React.PropTypes.string,
}

// Prop 预设值，若对应 props 没传入值将会使用 default 值
MyComponent.defaultProps = {
 name: '',
}

// 将 <MyComponent /> 组件插入 id 为 app 的 DOM 元素中
ReactDOM.render(<MyComponent name="Mark"/>, document.getElmentById('app'));
```

值得留意的是在 ES6 Class 中 `render()` 是唯一必要的方法（但要注意的是请保持 `redner()` 的纯粹，不要在里面进行 `state` 修改或是使用非同步方法和浏览器互动，若需非同步互动请于 `componentDidMount()` 操作），而 Functional Component 目前允许 `return null` 值。 喔对了，在 ES6 中也不支持 `mixins` 复用其他组件的方法了。

React Component 生命周期

React Component，就像人会有生老病死一样有生命周期。一般而言 Component 有以下三种生命周期的状态：

1. Mounting：已插入真实的 DOM
2. Updating：正在被重新渲染
3. Unmounting：已移出真实的 DOM

针对 Component 的生命周期状态 React 也有提供对应的处理方法：

1. Mounting

    - componentWillMount()
    - componentDidMount()
2. Updating
    - componentWillReceiveProps(object nextProps)：已载入组件收到新的参数时呼叫
    - shouldComponentUpdate(object nextProps, object nextState)：组件判断是否重新渲染时呼叫，起始不会呼叫除非呼叫 forceUpdate()
    - componentWillUpdate(object nextProps, object nextState)
    - componentDidUpdate(object prevProps, object prevState)
3. Unmounting
    - componentWillUnmount()

很多读者一开始学习 Component 生命周期时会觉得很抽象，所以接下来用一个简单范例让大家感受一下 Component 的生命周期。读者可以发现当一开始载入组件时第一个会触发 `console.log('constructor');` ，依序执行 `componentWillMount` 、 `componentDidMount` ，而当点击文字触发 `handleClick()` 更新 `state` 时则会依序执行 `componentWillUpdate` 、 `componentDidUpdate` ：

HTML Markup：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <script src="https://fb.me/react-15.1.0.js"></script>
  <script src="https://fb.me/react-dom-15.1.0.js"></script>
  <title>Component LifeCycle</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

Component 生命周期展示：

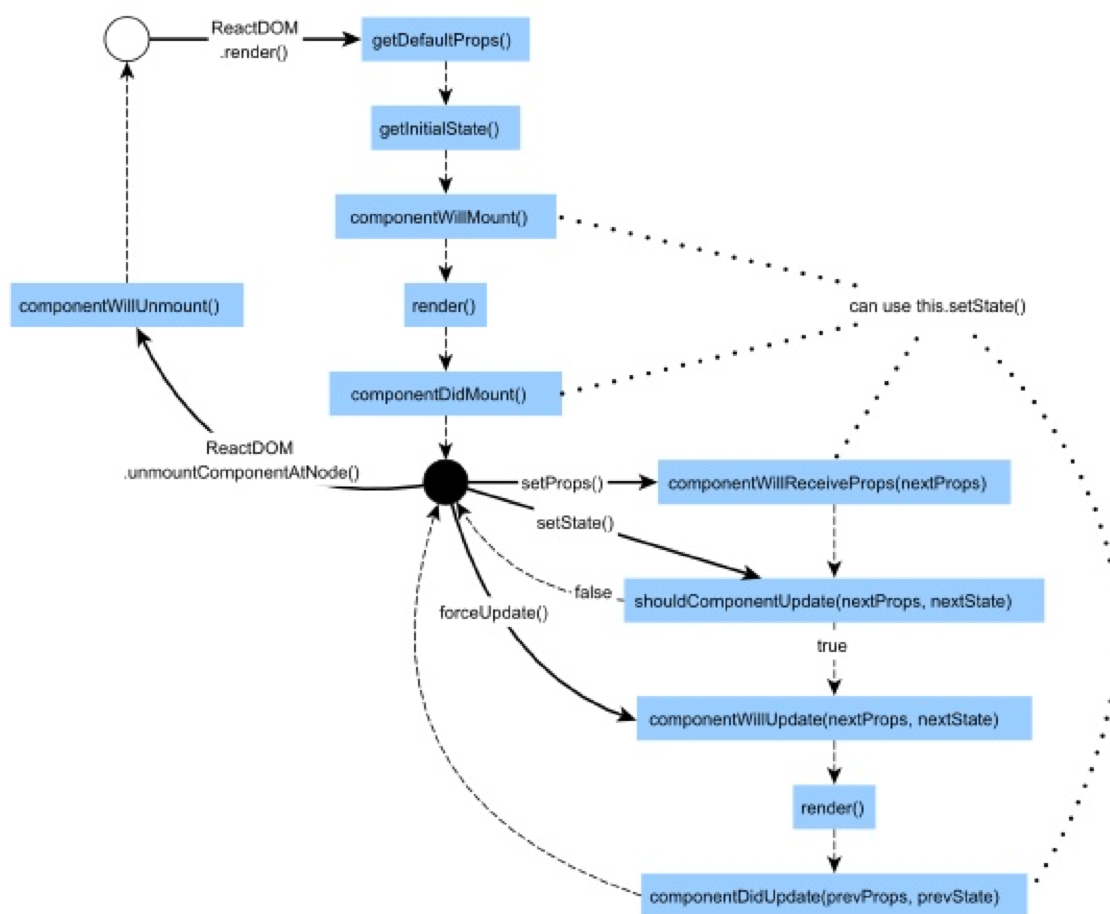
```
class MyComponent extends React.Component {
```

```
  constructor(props) {
    super(props);
    console.log('constructor');
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      name: 'Mark',
    }
  }
  handleClick() {
    this.setState({'name': 'Zuck'});
  }
  componentWillMount() {
    console.log('componentWillMount');
  }
  componentDidMount() {
    console.log('componentDidMount');
  }
  componentWillReceiveProps() {
    console.log('componentWillReceiveProps');
  }
  componentWillUpdate() {
    console.log('componentWillUpdate');
  }
  componentDidUpdate() {
    console.log('componentDidUpdate');
  }
  componentWillUnmount() {
    console.log('componentWillUnmount');
  }
  render() {
    return (
      <div onClick={this.handleClick}>Hi, {this.state.name}</div>

    );
  }
}

ReactDOM.render(<MyComponent />, document.getElementById('app'))
;
```

点击看详细范例



其中特殊处理的函数 `shouldComponentUpdate` ，目前预设 `return true` 。若你想要优化效能可以自己编写判断方式，若采用 `immutable` 可以使用 `nextProps === this.props` 比对是否有变动：

```
shouldComponentUpdate(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

Ajax 非同步处理

若有需要进行 Ajax 非同步处理，请在 `componentDidMount` 进行处理。以下通过 `jQuery` 执行 `Ajax` 取得 `Github API` 资料当做范例：

HTML Markup：

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8">
  <meta name="viewport" content="width=device-width">
  <script src="https://fb.me/react-15.1.0.js"></script>
  <script src="https://fb.me/react-dom-15.1.0.js"></script>
  <script src="https://code.jquery.com/jquery-3.1.0.js"></script>

  <title>GitHub User</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

app.js

```
class UserGithub extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
          username: '',
          githubtUrl: '',
          avatarUrl: '',
        }
    }
    componentDidMount() {
        $.get(this.props.source, (result) => {
            console.log(result);
            const data = result;
            if (data) {
              this.setState({
                    username: data.name,
                    githubtUrl: data.html_url,
                    avatarUrl: data.avatar_url
              });
            }
        });
    }
    render() {
        return (
          <div>
            <h3>{this.state.username}</h3>
            <img src={this.state.avatarUrl} />
            <a href={this.state.githubtUrl}>Github Link</a>.
          </div>
        );
    }
}

ReactDOM.render(
  <UserGithub source="https://api.github.com/users/torvalds" />,
  document.getElementById('app')
);
```

点击看详细范例

总结

以上介绍了 React Component 规格与生命周期（Life Cycle）的概念，其中生命周期的概念对于初学者来说可能会比较抽象，建议读者跟著范例动手实践。接下来我们将更进一步介绍 `React Router` 让读者感受一下单页式应用程序（single page application）的设计方式。

延伸阅读

1. Component Specs and Lifecycle

（image via react-lifecycle）

:door: 任意门

| 回首页 | 上一章：Props、State、Refs 与表单处理 | 下一章：React Router 入门实战教学 |

| 纠错、提问或许愿 |

Ch05 React Router

1. React Router 入门实战教学

:door: 任意门

| 回首页 |

React Router 入门实战教学



前言

若你是从一开始一路走到这里读者请先给自己一个爱的鼓励吧！在经历了 React 基础的训练后，相信各位读者应该都等不及想大展拳脚了！接下来我们将进行比较复杂的应用程序开发并和读者介绍目前市场上常见的不刷页单页式应用程序（single page application）的设计方式。

单页式应用程序（single page application）

传统的 Web 开发主要是由伺服器管理 URL Routing 和渲染 HTML 页面，过往每次 URL 一换或使用者连接一点，就需要重新从伺服器端重新载入页面。但随著使用者对于使用者体验的要求提升，许多的网页应用程序纷纷设计成不刷页的单页式应用程序（single page application），由前端负责 URL 的 routing 管理，若需要和后端进行 API 资料沟通的话，通常也会使用 Ajax 的技术。在 React 开发世界中主流是使用 react-router 这个 routing 管理用的 library。

React Router 环境设置

先透过以下指令在根目录产生 npm 配置文件 `package.json` ：

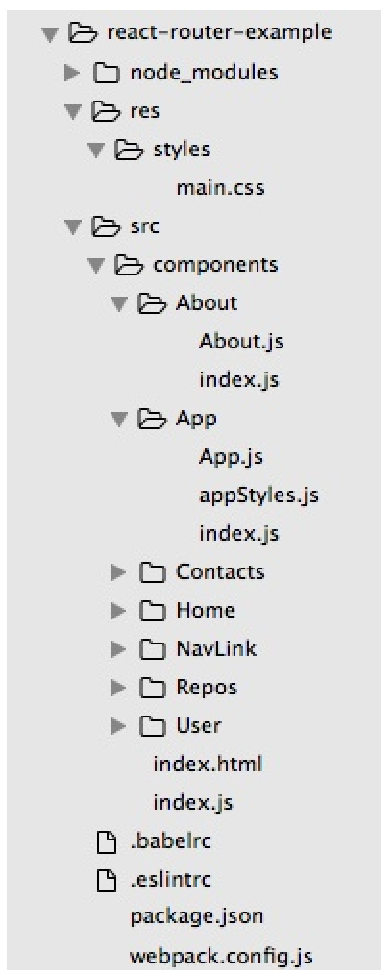
```
$ npm init
```

安装相关包（包含开发环境使用的包）：

```
$ npm install --save react react-dom react-router
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader babel-preset-es2015 babel-preset-react eslint eslint-config-airbnb eslint-loader eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react webpack webpack-dev-server html-webpack-plugin
```

安装好后我们可以设计一下我们的文件夹结构，首先我们在根目录建立 `src` 和 `res` 文件夹，分别放置 `script` 的 `source` 和静态资源（如：全域使用的 `.css` 和图档）。在 `components` 文件夹中我们会放置所有 `components`（个别组件文件夹中会用 `index.js` 输出组件，让引入组件更简洁），其余配置文件则放置于根目录下。



接下来我们先设定一下开发文档。

1. 设定 Babel 的配置文件： `.babelrc`

```
{
    "presets": [
      "es2015",
      "react",
    ],
    "plugins": []
}
```

2. 设定 ESLint 的配置文件和规则： `.eslintrc`

```
{
  "extends": "airbnb",
  "rules": {
    "react/jsx-filename-extension": [1, { "extensions": [".
js", ".jsx"] }],
  },
  "env" :{
    "browser": true,
  }
}
```

3. 设定 Webpack 配置文件： `webpack.config.js`

```
// 让你可以动态插入 bundle 好的 .js 档到 .index.html
const HtmlWebpackPlugin = require('html-webpack-plugin');

const HTMLWebpackPluginConfig = new HtmlWebpackPlugin({
  template: `${__dirname}/src/index.html`,
  filename: 'index.html',
  inject: 'body',
});

// entry 为进入点，output 为进行完 eslint、babel loader 转译后的
档案位置
module.exports = {
  entry: [
```

```
      './src/index.js',
    ],
    output: {
      path: `${__dirname}/dist`,
      filename: 'index_bundle.js',
    },
    module: {
      preLoaders: [
        {
          test: /\.jsx$|\.js$/,
          loader: 'eslint-loader',
          include: `${__dirname}/src`,
          exclude: /bundle\.js$/
        }
      ],
      loaders: [{
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react'],
        },
      }],
    },
    // 启动开发测试用 server 设定（不能用在 production）
    devServer: {
      inline: true,
      port: 8008,
    },
    plugins: [HTMLWebpackPluginConfig],
  };
```

太好了！这样我们就完成了开发环境的设定可以开始动手实践 `React Router` 应用程序了！

开始 React Routing 之旅

HTML Markup：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>ReactRouter</title>
  <link rel="stylesheet" type="text/css" href="../res/styles/mai
n.css">
</head>
<body>
    <div id="app"></div>
</body>
</html>
```

以下是 `webpack.config.js` 的进入点 `src/index.js` ，负责管理 `Router` 和 `render` 组件。这边我们要先详细讨论的是，为了使用 React Router 功能引入了许多 `react-router` 内部的组件。

1. Router `Router` 是放置 Route 的容器，其本身不定义 routing ，真正 routing 规则由 `Route` 定义。

2. Route `Route` 负责 URL 和对应的组件关系，可以有多个 `Route` 规则也可以有嵌套（nested） `Routing` 。像下面的例子就是每个页面都会先载入 `App` 组件再载入对应 URL 的组件。

3. history `Router` 中有一个属性 `history` 的规则，这边使用我们使用 `hashHistory` ，使用 routing 将由 `hash` （#）变化决定。例如：当使用者拜访 `http://www.github.com/` ，实际看到的会是 `http://www.github.com/#/` 。下列范例若是拜访了 `/about` 则会看到 `http://localhost:8008/#/about` 并载入 `App` 组件再载入 `About` 组件。

    - hashHistory 教学范例使用的，会通过 `hash` 进行对应。好处是简单易用，不用多余设定。

    - browserHistory 适用于伺服器端渲染，但需要设定伺服器端避免处理错误，这部份我们会在后面的章节详细说明。注意的是若是使用 Webpack 开发用伺服器需加上 `--history-api-fallback`

```
$ webpack-dev-server --inline --content-base . --history-api-fallback
```

- createMemoryHistory 主要用于伺服器渲染，使用上会建立一个存在记忆体的 `history` 物件，不会修改浏览器的网址位置。

```
const history = createMemoryHistory(location)
```

4. path `path` 是对应 URL 的规则。例如：`/repos/torvalds` 会对应到 `/repos/:name` 的位置，并将参数传入 `Repos` 组件中。由 `this.props.params.name` 取得参数。顺带一提，若为查询参数 `/user?q=torvalds` 则由 `this.props.location.query.q` 取得参数。
5. IndexRoute 由于 `/` 情况下 App 组件对应的 `this.props.children` 会是 `undefinded`，所以使用 `IndexRoute` 来解决对应问题。这样当 URL 为 `/` 时将会对应到 Home 组件。不过要注意的是 `IndexRoute` 没有 path 属性。

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Router, Route, hashHistory, IndexRoute } from 'react-ro
uter';
import App from './components/App';
import Home from './components/Home';
import Repos from './components/Repos';
import About from './components/About';
import User from './components/User';
import Contacts from './components/Contacts';

ReactDOM.render(
  <Router history={hashHistory}>
    <Route path="/" component={App}>
      <IndexRoute component={Home} />
      <Route path="/repos/:name" component={Repos} />
      <Route path="/about" component={About} />
      <Route path="/user" component={User} />
      <Route path="/contacts" component={Contacts} />
    </Route>
  </Router>,
  document.getElementById('app'));

  /* 另外一种写法：
    const routes = (
        <Route path="/" component={App}>
          <IndexRoute component={Home} />
          <Route path="/repos/:name" component={Repos} />
          <Route path="/about" component={About} />
          <Route path="/user" component={User} />
          <Route path="/contacts" component={Contacts} />
        </Route>
    );

    ReactDOM.render(
      <Router routes={routes} history={hashHistory} />,
      document.getElementById('app'));
  */
```

由于我们在 `index.js` 使用嵌套 routing，把 App 组件当做每个组件都会载入的母模版，亦即进入每个对应页面载入对应组件前都会先载入 App 组件。这样就可以让每个页面都有导览列连接可以点选，同时可以透过 `props.children` 载入对应 URL 的子组件。

1. Link `Link` 组件主要用于点击后连接转换，可以想成是 `<a>` 超连接的 React 版本。若是希望当点击时候有对应的 css style，可以使用 `activeStyle` 、 `activeClassName` 去做设定。范例分别使用于 `index.html` 使用传统 `CSS` 载入、Inline Style、外部引入 `Inline Style` 写法。
2. IndexLink IndexLink 主要是了处理 `index` 用途，特别注意当 child route `actived` 时，parent route 也会 `actived` 。所以我们回首页的连接使用 `<IndexLink />` 内部的 `onlyActiveOnIndex` 属性来解决这个问题。
3. Redirect、IndexRedirect 这边虽然没有用到，但若读者有需要使用到连接跳转的话可以参考这两个组件，用法类似于 `Route` 和 `IndexRedirect` 。

以下是 `src/components/App/App.js` 完整代码：

```
import React from 'react';
import { Link, IndexLink } from 'react-router';
import styles from './appStyles';
import NavLink from '../NavLink';

const App = (props) => (
  <div>
    <h1>React Router Tutorial</h1>
    <ul>
      <li><IndexLink to="/" activeClassName="active">Home</IndexLink></li>
      <li><Link to="/about" activeStyle={{ color: 'green' }}>About</Link></li>
      <li><Link to="/repos/react-router" activeStyle={styles.active}>Repos</Link></li>
      <li><Link to="/user" activeClassName="active">User</Link></li>
      <li><NavLink to="/contacts">Contacts</NavLink></li>
    </ul>
    <!-- 我们将 App 组件当做每个组件都会载入的母模版，因此可以透过 children 载入对应 URL 的子组件 -->
    {props.children}
  </div>
);

App.propTypes = {
  children: React.PropTypes.object,
};

export default App;
```

对应的组件内部使用 Functional Component 进行 UI 渲染：

以下是 `src/components/Repos/Repos.js` 完整代码：

```
import React from 'react';

const Repos = (props) => (
  <div>
    <h3>Repos</h3>
    <h5>{props.params.name}</h5>
  </div>
);

Repos.propTypes = {
  params: React.PropTypes.object,
};

export default Repos;
```

详细的代码读者可以参考范例文件夹，若读者跟著范例完成的话，可以在终端机上执行 `npm start` ，并于浏览器 `http://localhost:8008` 看到以下成果，当你点选连接时会切换对应组件并改变 `actived` 状态！

React Router Tutorial

- Home
- About
- Repos
- User
- Contacts

Repos

react-router

总结

到这边我们又一起完成了一个重要的一关，学习 `routing` 对于使用 `React` 开发复杂应用程序是非常重要的一步，接下来我们将一起学习一个相对独立的单元 `ImmutableJS` ，但学习 `ImmutableJS` 可以让我们在使用 `React` 和 `Flux/Redux` 可以有更好的性能和避免一些副作用。

延伸阅读

1. Leveling Up With React: React Router
2. Programmatically navigate using react router
3. React Router 使用教程
4. React Router 中文文档
5. React Router Tutorial

（iamge via seanamarasinghe）

任意门

| 回首页 | 上一章：React Component 规格与生命周期（Life Cycle） | 下一章：ImmutableJS 入门教学 |

| 纠错、提问或许愿 |

Ch06 ImmutableJS

:door: 任意门

| 回首页 |

ImmutableJS 入门教学



前言

一般来说在 JavaScript 中有两种数据类型：Primitive（String、Number、Boolean、null、undefinded）和 Object（Reference）。在 JavaScript 中对象的操作比起 Java 容易很多，但也因为相对弹性不严谨，所以产生了一些问题。在 JavaScript 中的 Object（对象）数据是 Mutable（可以变的），由于是使用 Reference 的方式，所以当修改到复制的值也会修改到原始值。例如下面的 `map2` 值是指到 `map1`，所以当 `map1` 值一改，`map2` 的值也会受影响。

```
var map1 = { a: 1 };
var map2 = map1;
map2.a = 2
```

通常一般做法是使用 `deepCopy` 来避免修改，但这样做法会产生较多的资源浪费。为了很好的解决这个问题，我们可以使用 `Immutable Data`，所谓的 Immutable Data 就是一旦建立，就不能再被修改的数据数据。

为了解决这个问题，在 2013 年时 Facebook 工程师 Lee Byron 打造了 ImmutableJS，但并没有被预设放到 React 工具包中（虽然有提供简化的 Helper），但 `ImmutableJS` 的出现确实解决了 `React` 甚至 `Redux` 所遇到的一些问题。

以下范例即是引入了 `ImmutableJS` 的效果，读者可以发现，虽然我们操作了 `map1` 的值，但会发现原本的 `map1` 并未受到影响（因为任何修改都不会影响到原始数据），虽然使用 `deepCopy` 也可以模拟类似的效果但会浪费过多的计算资

源和内存， `ImmutableJS` 则可以容易地共享没有被修改到的数据（例如下面的数据 `b` 即为 `map1` 所 `map2` 共享），因而有更好的性能表现。

```
import Immutable from 'immutable';

var map1 = Immutable.Map({ a: 1, b: 3 });
var map2 = map1.set('a', 2);

map1.get('a'); // 1
map2.get('a'); // 2
```

ImmutableJS 特性介绍

ImmutableJS 提供了 7 种不可修改的数据类型： `List` 、 `Map` 、 `Stack` 、 `OrderedMap` 、 `Set` 、 `OrderedSet` 、 `Record` 。若是对 Immutable 对象操作都会回传一个新值。其中比较常用的有 `List` 、 `Map` 和 `Set` ：

1. Map：类似于 key/value 的 object，在 ES6 也有原生 `Map` 对应

```
const Map= Immutable.Map;

// 1. Map 大小
const map1 = Map({ a: 1 });
map1.size
// => 1

// 2. 新增或取代 Map 元素
// set(key: K, value: V)
const map2 = map1.set('a', 7);
// => Map { "a": 7 }

// 3. 删除元素
// delete(key: K)
const map3 = map1.delete('a');
// => Map {}

// 4. 清除 Map 内容
const map4 = map1.clear();
// => Map {}

// 5. 更新 Map 元素
// update(updater: (value: Map<K, V>) => Map<K, V>)
// update(key: K, updater: (value: V) => V)
// update(key: K, notSetValue: V, updater: (value: V) => V)
const map5 = map1.update('a', () => (7))
// => Map { "a": 7 }

// 6. 合并 Map
const map6 = Map({ b: 3 });
map1.merge(map6);
// => Map { "a": 1, "b": 3 }
```

2. List：有序且可以重复值，对应于一般的 Array

```
const List= Immutable.List;

// 1. 取得 List 长度
const arr1 = List([1, 2, 3]);
arr1.size
// => 3

// 2. 新增或取代 List 元素内容
// set(index: number, value: T)
// 将 index 位置的元素替换
const arr2 = arr1.set(-1, 7);
// => [1, 2, 7]
const arr3 = arr1.set(4, 0);
// => [1, 2, 3, undefined, 0]

// 3. 删除 List 元素
// delete(index: number)
// 删除 index 位置的元素
const arr4 = arr1.delete(1);
// => [1, 3]

// 4. 插入元素到 List
// insert(index: number, value: T)
// 在 index 位置插入 value
const arr5 = arr1.insert(1, 2);
// => [1, 2, 2, 3]

// 5. 清空 List
// clear()
const arr6 = arr1.clear();
// => []
```

3. Set：没有顺序且不能重复的列表

```
const Set= Immutable.Set;

// 1. 建立 Set
const set1 = Set([1, 2, 3]);
// => Set { 1, 2, 3 }

// 2. 新增元素
const set2 = set1.add(1).add(5);
// => Set { 1, 2, 3, 5 }
// 由于 Set 为不能重复集合，故 1 只能出现一次

// 3. 删除元素
const set3 = set1.delete(3);
// => Set { 1, 2 }

// 4. 取联集
const set4 = Set([2, 3, 4, 5, 6]);
set1.union(set4);
// => Set { 1, 2, 3, 4, 5, 6 }

// 5. 取交集
set1.intersect(set4);
// => Set { 2, 3 }

// 6. 取差集
set1.subtract(set4);
// => Set { 1 }
```

ImmutableJS 的特性整理

1. Persistent Data Structure 在 `ImmutableJS` 的世界里，只要数据一被创建，就不能修改，维持 `Immutable` 。就不会发生下列的状况：

```
var obj = {
a: 1
};

funcationA(obj);
console.log(obj.a) // 不确定结果为多少？
```

使用 `ImmutableJS` 就没有这个问题：

```
// 有些开发者在使用时会在 ``Immutable` 变数前加 `$` 以示区隔。

const $obj = fromJS({
a: 1
});

funcationA($obj);
console.log($obj.get('a')) // 1
```

2. Structural Sharing 为了维持数据的不可变，又要避免像 `deepCopy` 一样复制所有的节点数据而造成的资源损耗，在 `ImmutableJS` 使用的是 Structural Sharing 特性，亦即如果对象树中一个节点发生变化的话，只会修改这个节点和和受它影响的父节点，其他节点则共享。

```
const obj = {
 count: 1,
 list: [1, 2, 3, 4, 5]
}
var map1 = Immutable.fromJS(obj);
var map2 = map1.set('count', 4);

console.log(map1.list === map2.list); // true
```

3. Support Lazy Operation

```
Immutable.Range(1, Infinity)
.map(n => -n)
// Error: Cannot perform this action with an infinite size.

Immutable.Range(1, Infinity)
.map(n => -n)
.take(2)
.reduce((r, n) => r + n, 0);
// -3
```

4. 丰富的 API 并提供快速转换原生 JavaScript 的方式 在 ImmutableJS 中可以使用 `fromJS()` 、 `toJS()` 进行 JavaScript 和 ImmutableJS 之间的转换。但由于在转换之间会非常耗费资源，所以若是你决定引入 `ImmutableJS` 的话请尽量维持数据处在 `Immutable` 的状态。
5. 支持 Functional Programming `Immutable` 本身就是 Functional Programming（函数式程式设计）的概念，所以在 `ImmutableJS` 中可以使用许多 Functional Programming 的方法，例如： `map` 、 `filter` 、 `groupBy` 、 `reduce` 、 `find` 、 `findIndex` 等。
6. 容易实现 Redo/Undo 历史回顾

React 性能优化

`ImmutableJS` 除了可以和 `Flux/Redux` 整合外，也可以用于基本 react 性能优化。以下是一般使用性能优化的简单方式：

传统 JavaScript 比较方式，若数据型态为 Primitive 就不会有问题：

```
// 在 shouldComponentUpdate 比较接下来的 props 一否一致，若相同则不重新渲染，提升性能
shouldComponentUpdate (nextProps) {
    return this.props.value !== nextProps.value;
}
```

但当比较的是对象的话就会出现问题：

```
// 假设 this.props.value 为 { foo: 'app' }
// 架设 nextProps.value 为 { foo: 'app' },
// 虽然两者值是一样，但由于 reference 位置不同，所以视为不同。但由于值一样应该要避免重复渲染
this.props.value !== nextProps.value; // true
```

使用 `ImmutableJS`：

```
var SomeRecord = Immutable.Record({ foo: null });
var x = new SomeRecord({ foo: 'app'  });
var y = x.set('foo', 'azz');
x === y; // false
```

在 ES6 中可以使用官方文件上的 `PureRenderMixin` 进行比较，可以让程式码更简洁：

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }
  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

总结

虽然 `ImmutableJS` 的引入可以带来许多好处和性能的提升但由于引入整体档案较大且较具侵入性，在引入之前可以自行评估看看是否合适于目前的专案。接下来我们将在后面的章节讲解如何将 `ImmutableJS` 和 `Redux` 整合应用到实务上的范例。

延伸阅读

1. 官方网站
2. Immutable.js初识
3. Immutable 详解及 React 中实践
4. 为什么需要Immutable.js
5. facebook immutable.js 意义何在，使用场景？
6. React 巢状 Component 性能优化
7. PureRenderMixin
8. seamless-immutable
9. Immutable Data Structures and JavaScript

（image via risingstack）

:door: 任意门

| 回首页 | 上一章：React Router 入门实战教学 | 下一章：Flux 基础概念与实战入门 |

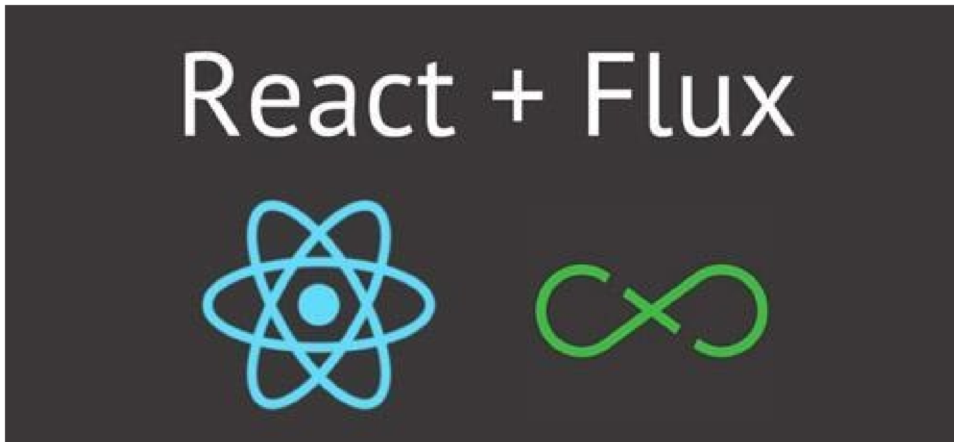
| 纠错、提问或许愿 |

Ch07 Flux/Redux

:door: 任意门

| 回首页 |

Flux 基础概念与实战入门

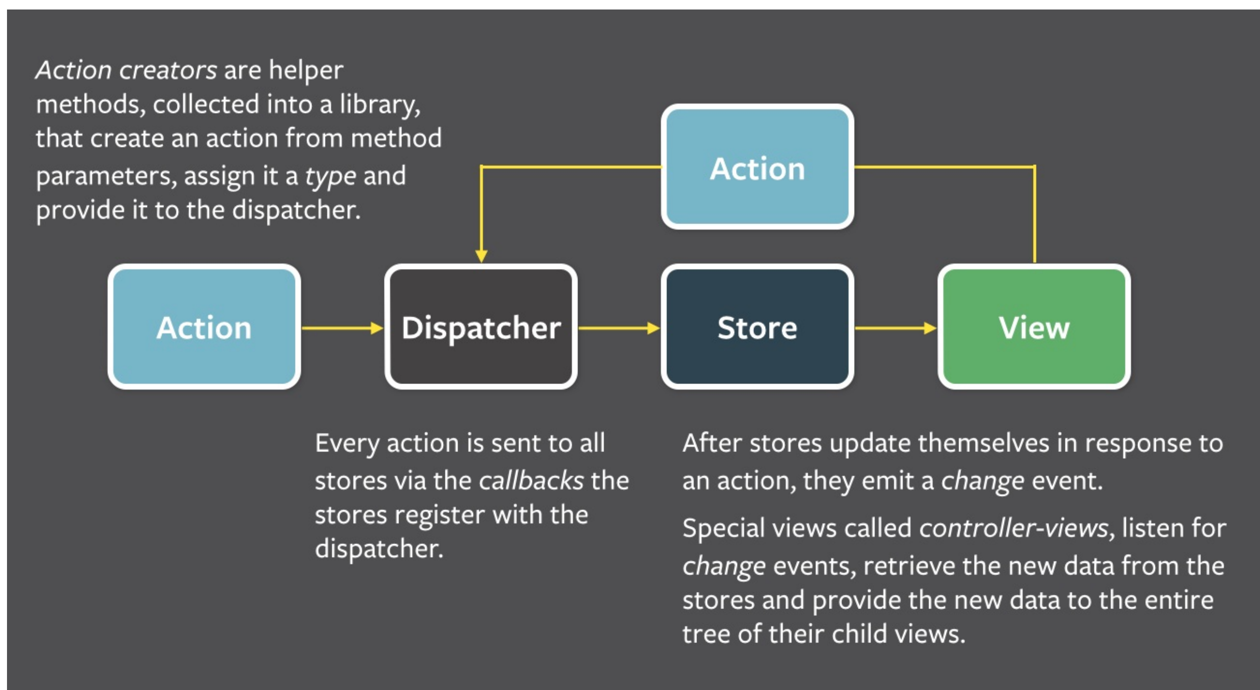


前言

随著 React App 复杂度提升，我们会发现常常需要从 Parent Component 通过 props 传递方法到 Child Component 去改变 state tree，不但不方便也难以管理，因此我们需要更好的数据架构来建置更复杂的应用程序。Flux 是 Facebook 推出的 client-side 应用程序架构（Architecture），主要想解决 `MVC` 架构的一些问题。事实上，Flux 并非一个完整的前端 Framework，其特色在于实现了 Unidirectional Data Flow（单向流）的数据流设计模式，在开发复杂的大型应用程序时可以更容易地管理 state（状态）。由于 React 主要是负责 View 的部份，所以通过搭配 Flux-like 的数据处理架构，可以更好的去管理我们的 state（状态），处理复杂的使用者互动（例如：Facebook 同时要维护使用者是否点赞、点击相片，是否有新讯息等状态）。

由于原始的 Flux 架构在实现上有些部分可以精简和改善，在实际操作上我们通常会使用开发者社群开发的 Flux-like 相关的架构实现（例如：Redux、Alt、Reflux 等）。不过这边我们主要会使用 Facebook 本身提供 `Dispatcher API` 函式库（可以想成是一个 pub/sub 处理器，通过 broadcast 将 `payloads` 传给注册的 callback function）并搭配 `NodeJS` 的 `EventEmitter` 模块去完成 Flux 架构的实现。

Flux 概念介绍



在 Flux Unidirectional Data Flow（单项流）世界里有四大主角，分别负责不同对应的工作：

1. actions / Action Creator

   action 负责定义所有改变 state（状态）的行为，可以让开发者快速了解 App 的各种功能，若你想改变 state 你只能发 action。注意 action 可以是同步或是非同步。例如：新增代办事项，调用非同步 API 获取数据。

   实际操作上我们会分成 action 和 Action Creator。action 为描述行为的 object（物件），Action Creator 将 action 送给 dispatcher。一般来说符合 Flux Standard Action 的 action 会如以下范例代码，具备 `type` 来区别所触发的行为。而 `payload` 则是所夹带的数据：

```
// action
const addTodo = {
 type: 'ADD_TODO',
 payload: {
   text: 'Do something.'
 }
}

AppDispatcher.dispatch(addTodo);
```

   当发生 rejected Promise 情况：

```
{
 type: 'ADD_TODO',
 payload: new Error(),
 error: true
}
```

2. Dispatcher

   `Dispatcher` 是 Flux 架构的核心，每个 App 只有一个 Dispatcher，提供 API 让 store 可以注册 `callback function` ，并负责向所有 store 发送 action 事件。在本范例中我们使用 Facebook 提供的 Dispatcher API，其内建有 `dispatch` 和 `subscribe` 方法。

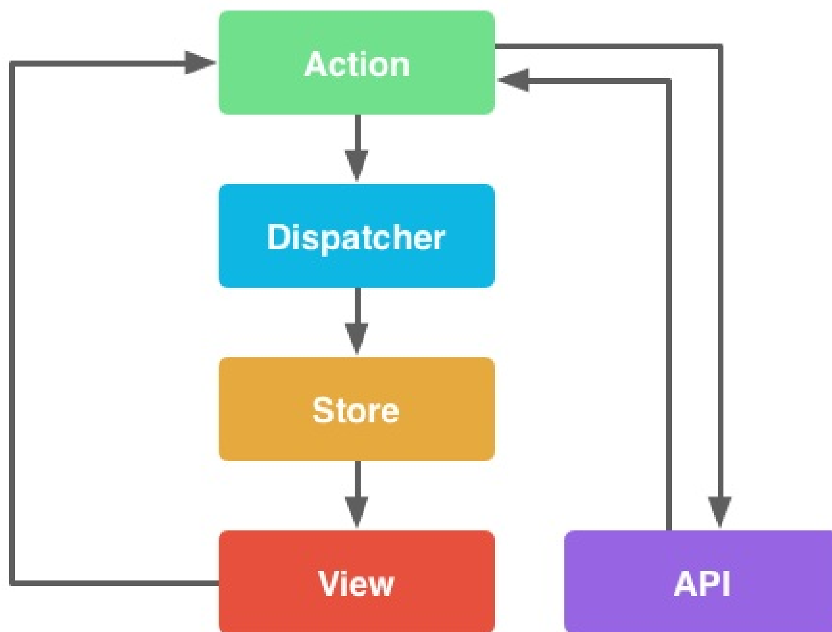
3. Stores

   一个 App 通常会有多个 store 负责存放业务逻辑，根据不同业务会有不同 store，例如：TodoStore、RecipeStore。 store 负责操作和储存数据并提供 `view` 使用 `listener` （监听器），若有数据更新即会触发更新。值得注意的是 store 只提供 `getter API` 读取数据，若想改变 state 一律发送 action。

4. Views（Controller Views）

   这部份是 `React` 负责的范畴，负责提供监听事件的 `callback function` ，当事件发生时重新取得数据并重绘 `View` 。

Flux 流程回顾



Flux 架构前置作业：

1. Stores 向 Dispatcher 注册 callback，当数据改变时告知 Stores
2. Controller Views 向 Stores 取得初始数据
3. Controller Views 将数据给 Views 去渲染 UI
4. Controller Views 向 store 注册 listener，当数据改变时告知 Controller Views

Flux 与使用者互动运作流程：

1. 使用者和 App 互动，触发事件，Action Creator 发送 actions 给 Dispatcher
2. Dispatcher 依序将 action 传给 store 并由 action type 判断合适的处理方式
3. 若有数据更新则会触发 Controller Views 向 store 注册的 listener 并向 store 取得更新数据
4. View 根据 Controller Views 的新数据重新绘制 UI

Flux 实战初体验

介绍完了整个 Flux 基本架构后，接下来我们就来动手实践一个简单 Flux 架构的 Todo，让使用者可以在 `input` 输入代办事项并新增。

首先，我们先完成一些开发的前置作业，先通过以下指令在根目录产生 npm 配置文件 `package.json` ：

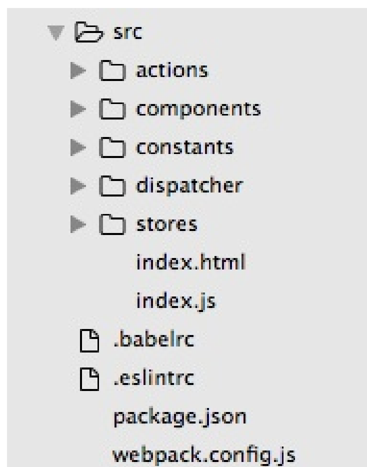
```
$ npm init
```

安装相关包（包含开发环境使用的包）：

```
$ npm install --save react react-dom flux events
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader babel-preset-es2015 babel-preset-react eslint eslint-config-airbnb eslint-loader eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react html-webpack-plugin webpack webpack-dev-server
```

安装好后我们可以设计一下我们的文件夹结构，首先我们在根目录建立 `src`，放置 `script` 的 `source`。在 `components` 文件夹中我们会放置所有 `components`（个别元件文件夹中会用 `index.js` 输出元件，让引入元件更简洁），另外还有 `actions`、`constants`、`dispatcher`、`stores`，其余配置文件则放置于根目录下。



接下来我们参考上一章设定一下开发文档（`.babelrc`、`.eslintrc`、`webpack.config.js`）。这样我们就完成了开发环境的设定可以开始动手实践 `React Flux` 应用程序了！

HTML Markup：

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>TodoFlux</title>
</head>
<body>
    <div id="app"></div>
</body>
</html>
```

以下为 `src/index.js` 完整代码，安排了父 `component` 和在 HTML Markup 插入位置：

```
import React from 'react';
import ReactDOM from 'react-dom';
import TodoHeader from './components/TodoHeader';
import TodoList from './components/TodoList';

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {};
  }
  render() {
    return (
      <div>
        <TodoHeader />
        <TodoList />
      </div>
    );
  }
}

ReactDOM.render(<App />, document.getElementById('app'));
```

通常实际操作上我们会开一个 `constants` 文件夹存放 `config` 或是 `actionTypes` 常数。以下是 `src/constants/actionTypes.js` ：

```
export const ADD_TODO = 'ADD_TODO';
```

在这个范例中我们继承了 Facebook 提供的 Dispatcher API（主要是继承了 `dispatch` 、 `register` 和 `subscribe` 的方法），打造自己的 DispatcherClass，当使用者触发 `handleAction()` 会 `dispatch` 出事件。以下是 `src/dispatch/AppDispatcher.js` ：

```
// Todo app dispatcher with actions responding to both
// view and server actions
import { Dispatcher } from 'flux';

class DispatcherClass extends Dispatcher {
  handleAction(action) {
    this.dispatch({
      type: action.type,
      payload: action.payload,
    });
  }
}

const AppDispatcher = new DispatcherClass();

export default AppDispatcher;
```

以下是我们利用 `AppDispatcher` 打造的 `Action Creator` 由 `handleAction` 负责发出传入的 `action` ，完整代码如 `src/actions/todoActions.js` ：

```
import AppDispatcher from '../dispatcher/AppDispatcher';
import { ADD_TODO } from '../constants/actionTypes';

export const TodoActions = {
  addTodo(text) {
    AppDispatcher.handleAction({
      type: ADD_TODO,
      payload: {
        text,
      },
    });
  },
};
```

`Store` 主要是负责数据以及业务逻辑处理，我们继承了 `events` 模块的 `EventEmitter` ，当 `action` 传入 `AppDispatcher.register` 的处理范围后，根据 `action type` 选择适合处理的 `store` 进行处理，处理完后通过 `emit` 方法发出事件让监听的 `Views Controller` 知道。以下是 `src/stores/TodoStore.js` ：

```
import AppDispatcher from '../dispatcher/AppDispatcher';
import { ADD_TODO } from '../constants/actionTypes';
import { EventEmitter } from 'events';

const store = {
  todos: [],
  editing: false,
};

class TodoStoreClass extends EventEmitter {
  addChangeListener(callback) {
    this.on(ADD_TODO, callback);
  }
  removeChangeListener(callback) {
    this.removeListener(ADD_TODO, callback);
  }
  getTodos() {
    return store.todos;
  }
}

const TodoStore = new TodoStoreClass();

AppDispatcher.register((action) => {
  switch (action.type) {
    case ADD_TODO:
      store.todos.push(action.payload.text);
      TodoStore.emit(ADD_TODO);
      break;
    default:
      return true;
  }
  return true;
});

export default TodoStore;
```

在这个 React Flux 范例中我们把 `View` 和 `Views Controller` 整合在一起。在 `TodoHeader` 中，我们主要任务是让使用者可以通过 `input` 新增代办事项。使用者输入文字在 `input` 时会触发 `onChange` 事件，进而更新内部的 `state`，当使用者按了送出钮就会触发 `onAdd` 事件，`dispatch` 出 `addTodo event`。以下是 `src/components/TodoHeader.js` 完整范例：

```
import React, { Component } from 'react';
import { TodoActions } from '../../actions/todoActions';

class TodoHeader extends Component {
  constructor(props) {
    super(props);
    this.onChange = this.onChange.bind(this);
    this.onAdd = this.onAdd.bind(this);
    this.state = {
      text: '',
      editing: false,
    };
  }
  onChange(event) {
    this.setState({
      text: event.target.value,
    });
  }
  onAdd() {
    TodoActions.addTodo(this.state.text);
    this.setState({
      text: '',
    });
  }
  render() {
    return (
      <div>
        <h1>TodoFlux</h1>
        <div>
          <input
            value={this.state.text}
            type="text"
            placeholder="请输入代办事项"
```

```
            onChange={this.onChange}
          />
          <button
            onClick={this.onAdd}
          >
            送出
          </button>
        </div>
      </div>
    );
  }
}

export default TodoHeader;
```

在上面的 Component 中我们让使用者可以新增代办事项，接下来我们要让新增的代办事项可以显示。我们在 `componentDidMount` 设了一个监听器 `TodoStore` 数据改变时会去把数据重新再更新，这样当使用者新增代办事项时 `TodoList` 就会保持同步。以下是 `src/components/TodoList.js` 完整代码：

```
import React, { Component } from 'react';
import TodoStore from '../../stores/TodoStore';

function getAppState() {
  return {
    todos: TodoStore.getTodos(),
  };
}
class TodoList extends Component {
  constructor(props) {
    super(props);
    this.onChange = this.onChange.bind(this);
    this.state = {
      todos: [],
    };
  }
  componentDidMount() {
    TodoStore.addChangeListener(this.onChange);
  }
  onChange() {
    this.setState(getAppState());
  }
  render() {
    return (
      <div>
        <ul>
          {
            this.state.todos.map((todo, key) => (
              <li key={key}>{todo}</li>
            ))
          }
        </ul>
      </div>
    );
  }
}

export default TodoList;
```

若读者都有跟著上面的步骤走完的话，最后我们在终端机的根目录位置执行 `npm start` 就可以看到整个成果，YA！

TodoFlux

請輸入代辦事項 送出

- 寫書
- 寫程式

总结

Flux 优势：

1. 让开发者可以快速了解整个 App 中的行为
2. 数据和业务逻辑统一存放好管理
3. 让 View 单纯化只负责 UI 的排版不需负责 state 管理
4. 清楚的架构和分工对于复杂中大型应用程序易于维护和管理代码

Flux 劣势：

1. 代码上不够简洁
2. 对于简单小应用来说稍微复杂

以上就是 Flux 的实战入门，我知道一开始接触 Flux 的读者一定会觉得很抽象，有些读者甚至会觉得这个架构到底有什么好处（明明感觉没比 MVC 高明到哪去或是一点都不简洁），但如同上述优点所说 Flux 设计模式的优势在于清楚的架构和分工对于复杂中大型应用程序易于维护和管理代码。若还是不熟悉的读者可以跟著范例多动手，相信慢慢就可以体会 Flux 的特色。事实上，在开发社区中为了让 Flux 架构更加简洁，产生了许多 Flux-like 的架构和函式库，接下来将带读者们进入目前最热门的架构：`Redux` 。

延伸阅读

1. Getting To Know Flux, the React.js Architecture
2. Flux 官方网站
3. 从 Flux 与 MVC 的差异来简介 Flux
4. Flux Stores and ES6
5. React and Flux: Migrating to ES6 with Babel and ESLint
6. Building an ES6/JSX/React Flux App – Part 2 – The Flux
7. Question: How to choose between Redux's store and React's state? #1287
8. acdlite/flux-standard-action

（image via devjournal、facebook、scotch.io）

:door: 任意门

| 回首页 | 上一章：ImmutableJS 入门教学 | 下一章：Redux 基础概念 |

| 纠错、提问或许愿 |

Redux 基础概念

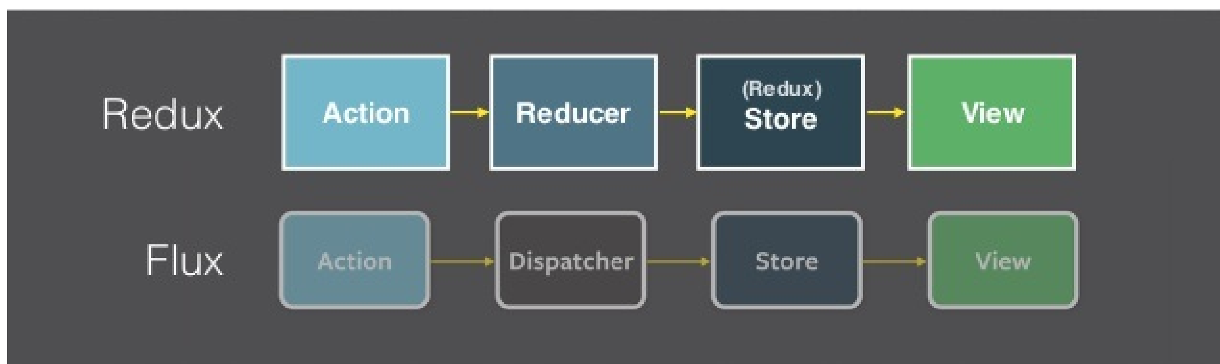


前言

前面一个章节我们讲解了 Flux 的功能和用法，但在实务上许多开发者较偏好的是同为 Flux-like 但较为简洁且文件丰富清楚的 Redux 当作状态数据管理的架构。Redux 是由 Dan Abramov 所发起的一个开源的 library，其主要功能如官方首页写着： `Redux is a predictable state container for JavaScript apps.` ，亦即 Redux 希望能提供一个可以预测的 state 管理容器，让开发者可以可以更容易开发复杂的 JavaScript 应用程序（注意 Redux 和 React 并无相依性，只是和 React 可以有很好的整合）。

Flux/Redux 超级比一比

从简单 Flux/Redux 比较图可以看出两者之间有些差异：



Redux vs traditional Flux

在开始实际操作 Redux App 之前我们先来了解一下 Redux 和 Flux 的一些差异：

1. 只使用一个 store 将整个应用程序的状态 (state) 用对象树 (object tree) 的方式储存起来：

   原生的 Flux 会有许多分散的 store 储存各个不同的状态，但在 redux 中，只会有唯一一个 store 将所有的数据用对象的方式包起来。

```
//原生 Flux 的 store
const userStore = {
    name: ''
}
const todoStore = {
    text: ''
}

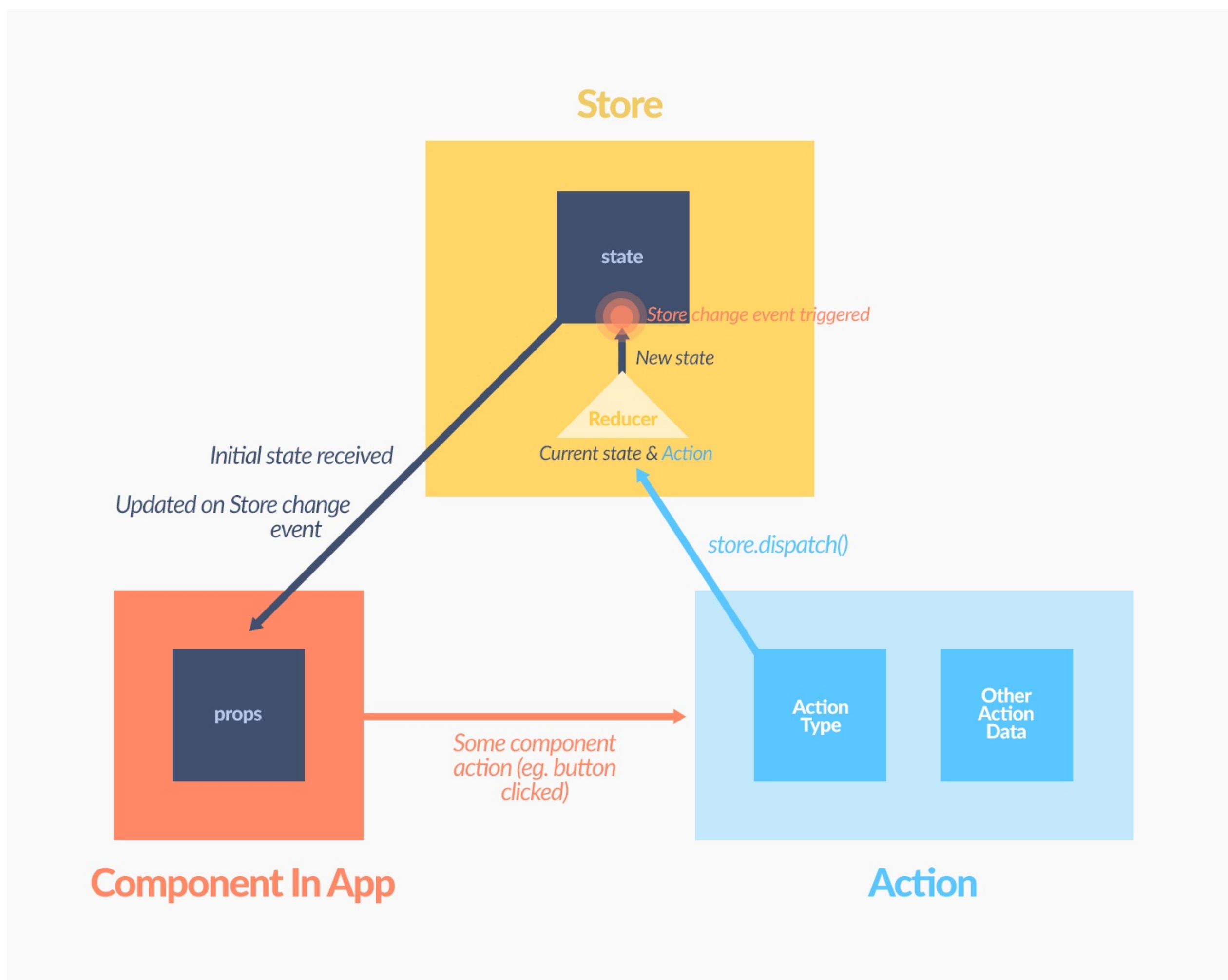
// Redux 的单一 store
const state = {
    userState: {
        name: ''
    },
    todoState: {
        text: ''
    }
}
```

2. 唯一可以改变 state 的方法就是发送 action，这部份和 Flux 类似，但 Redux 并没有像 Flux 设计有 Dispatcher。Redux 的 action 和 Flux 的 action 都是一个包含 `type` 和 `payload` 的对象。

3. Redux 拥有 Flux 所没有的 Reducer。Reducer 根据 action 的 type 去执行对应的 state 做变化的函数叫做 Reducer。你可以使用 switch 或是使用函数 mapping 的方式去对应处理的方式。

4. Redux 拥有许多方便好用的辅助测试工具（例如：redux-devtools、react-transform-boilerplate），方便测试和使用 `Hot Module Reload` 。

Redux 核心概念介绍



从上述的图中我们可以看到 Redux 数据流的模型大致上可以简化成： `View -> Action -> (Middleware) -> Reducer` 。当使用者和 View 互动时会触发事件发出 Action，若有使用 Middleware 的话会在进入 Reducer 进行一些处理，当 Action 进到 Reducer 时，Reducer 会根据，action type 去 mapping 对应处理的动作，然后回传回新的 state。View 则因为侦测到 state 更新而重绘页面。在这个章节我们讨论的是 synchronous（同步）的情形，asynchronous（非同步）的状况会在接下来的章节进行讨论。以下就用官方网站上的简单范例来让大家感受一下 Redux 的整个使用流程：

```
import { createStore } from 'redux';

/**
  下面是一个简单的 reducers ，主要功能是针对传进来的 action type 判断并回传新的 state
  reducer 规格：(state, action) => newState
  一般而言 state 可以是 primitive、array 或 object 甚至是 ImmutableJS Data。但要留意的是不能修改到原来的 state ，
```

```
  回传的是新的 state。由于使用在 Redux 中使用 ImmutableJS 有许多好处，
所以我们的范例 App 也会使用 ImmutableJS
*/
function counter(state = 0, action) {
  switch (action.type) {
  case 'INCREMENT':
    return state + 1;
  case 'DECREMENT':
    return state - 1;
  default:
    return state;
  }
}

// 创建 Redux store 去存放 App 的所有 state
// store 的可用 API { subscribe, dispatch, getState }
let store = createStore(counter);

// 可以使用 subscribe() 来订阅 state 是否更新。但实务通常会使用 react-r
edux 来串连 React 和 Redux
store.subscribe(() =>
  console.log(store.getState());
);

// 若想改变 state ，一律发 action
store.dispatch({ type: 'INCREMENT' });
// 1
store.dispatch({ type: 'INCREMENT' });
// 2
store.dispatch({ type: 'DECREMENT' });
// 1
```

Redux API 入门

1. createStore： `createStore(reducer, [preloadedState], [enhancer])`

   我们知道在 Redux 中只会有一个 store。在产生 store 时我们会使用 `createStore` 这个 API 来创建 store。第一个参数放入我们的 `reducer` 或是有多个 `reducers` combine（使用 `combineReducers` ）在一起的

`rootRuducers` 。第二个参数我们会放入希望预先载入的 `state` 例如：user session 等。第三个参数通常会放入我们想要使用用来增强 Redux 功能的 `middlewares` ，若有多个 `middlewares` 的话，通常会使用 `applyMiddleware` 来整合。

2. Store

   属于 Store 的四个方法：

   - getState()
   - dispatch(action)
   - subscribe(listener)
   - replaceReducer(nextReducer)

     关于 Store 重点是要知道 Redux 只有一个 Sotre 负责存放整个 App 的 State，而唯一能改变 State 的方法只有发送 action。

3. combineReducers： `combineReducers(reducers)`

   combineReducers 可以将多个 reducers 进行整合并回传一个 Function，让我们可以将 reducer 适度分割

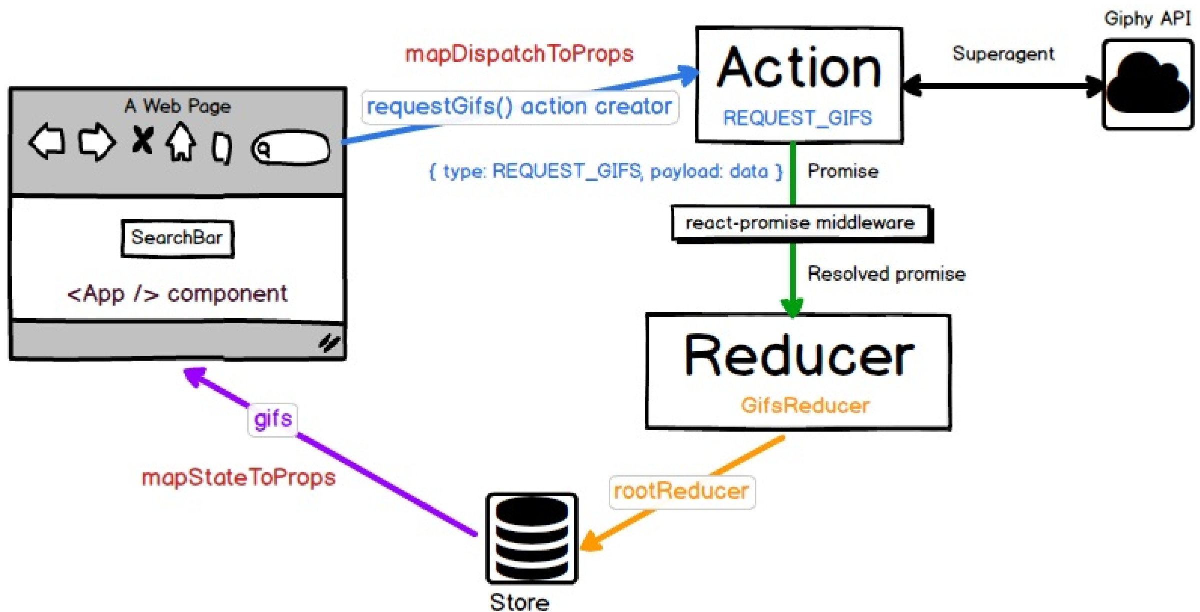
4. applyMiddleware： `applyMiddleware(...middlewares)`

   官方针对 Middleware 进行说明

   > It provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.

   若有 NodeJS 的经验的读者，对于 middleware 概念应该不陌生，让开发者可以在 req 和 res 之间进行一些操作。在 Redux 中 Middleware 则是扮演 action 到达 reducer 前的第三方扩充。而 applyMiddleware 可以将多个 `middlewares` 整合并回传一个 Function，便于使用。

   若是你要使用 asynchronous（非同步）的行为的话需要使用其中一种 middleware： redux-thunk、redux-promise 或 redux-promise-middleware ，这样可以让你在 actions 中 dispatch Promises 而非 function。asynchronous（非同步）运作方式就如同下图所示：



5. bindActionCreators： `bindActionCreators(actionCreators, dispatch)`

   bindActionCreators 可以将 `actionCreators` 和 `dispatch` 绑定，并回传一个 Function 或 Object，让程序更简洁。但若是使用 react-redux 可以用 `connect` 让 dispatch 行为更容易管理

6. compose： `compose(...functions)`

   compose 可以将 function 由右到左合并并回传一个 Function，如官网范例所示：

```
import { createStore, combineReducers, applyMiddleware, com
pose } from 'redux'
import thunk from 'redux-thunk'
import DevTools from './containers/DevTools'
import reducer from '../reducers/index'

const store = createStore(
  reducer,
  compose(
    applyMiddleware(thunk),
    DevTools.instrument()
  )
)
```

总结

以上介绍了 Redux 的基础概念，若是读者觉得还是有点抽象的话也没关系，在下一个章节我们将实际带大家开发一个整合 `React` 、 `Redux` 和 `ImmutableJS` 的 TodoApp。

延伸阅读

1. Redux 官方网站
2. Redux架构实践——Single Source of Truth
3. Presentational and Container Components
4. 使用Redux管理你的React应用
5. Using redux

（image via githubusercontent、makeitopen、css-tricks、tighten、tryolabs、facebook、JonasOhlsson）

:door: 任意门

| 回首页 | 上一章：Flux 基础概念与实战入门 | 下一章：Redux 实战入门 |

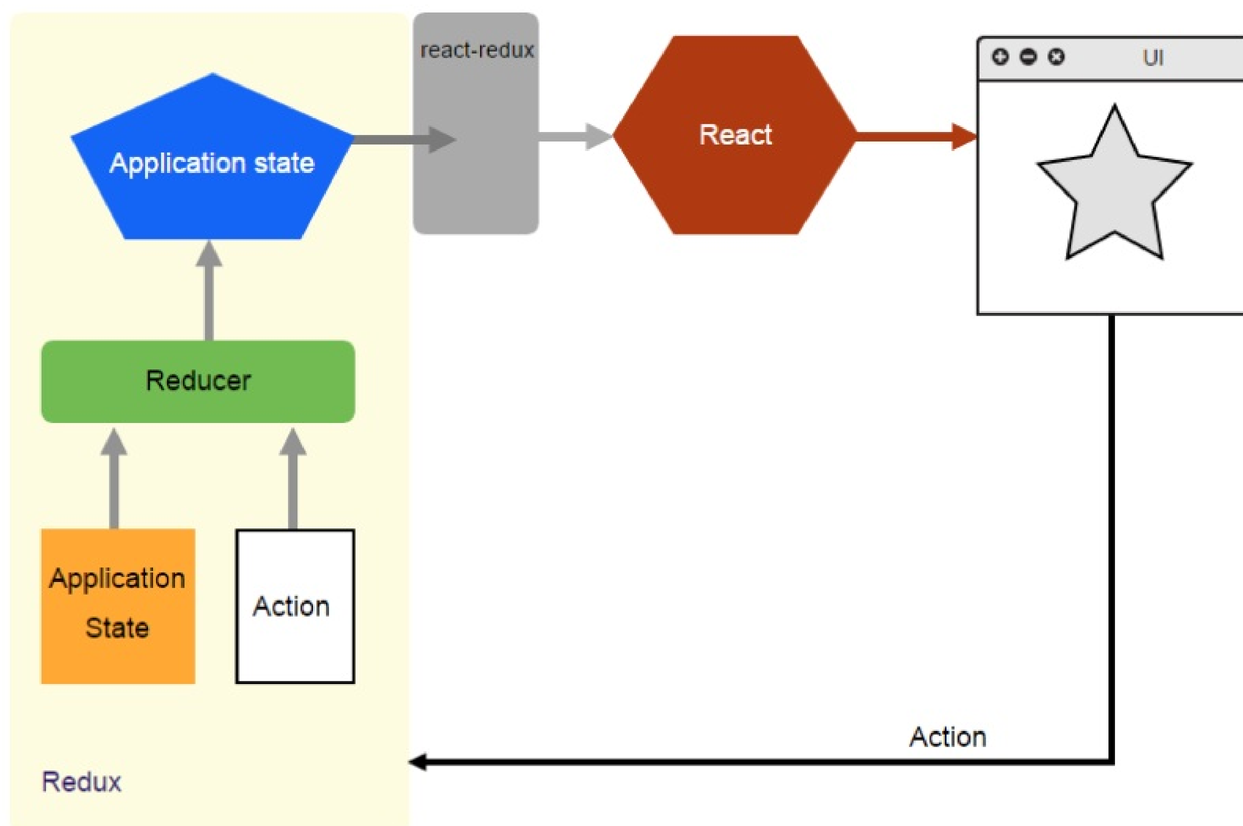
| 纠错、提问或许愿 |

Redux 实战入门

前言

上一节我们了解了 Redux 基本的概念和特性后，本章我们要实际动手用 Redux、React Redux 结合 ImmutableJS 开发一个简单的 Todo 应用。话不多说，那就让让我们开始吧！

以下这张图表示了整个 React Redux App 的数据流程图（使用者与 View 互动 => dispatch 出 Action => Reducers 依据 action type 分配到对应处理方式，回传新的 state => 通过 React Redux 传送给 React，React 重新绘制 View）：



动手创作 React Redux ImmutableJS TodoApp

在开始创作之前我们先完成一些开发的前置作业，先通过以下指令在根目录产生 npm 配置文件 `package.json` ：

```
$ npm init
```

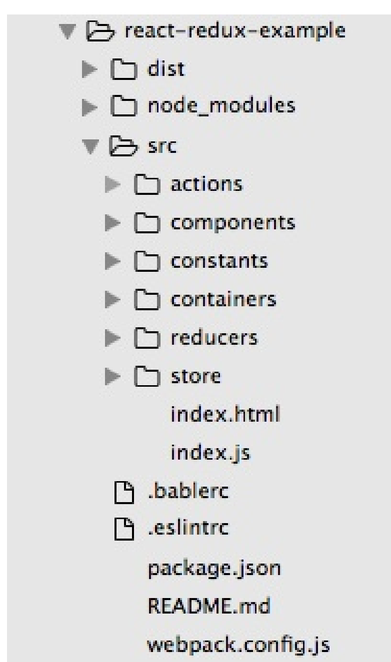
安装相关包（包含开发环境使用的包）：

```
$ npm install --save react react-dom redux react-redux immutable
 redux-actions redux-immutable
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader ba
bel-preset-es2015 babel-preset-react eslint eslint-config-airbnb
 eslint-loader eslint-plugin-import eslint-plugin-jsx-a11y eslin
t-plugin-react html-webpack-plugin webpack webpack-dev-server
```

安装好后我们可以设计一下我们的文件夹目录结构，首先我们在根目录建立 `src` ，放置 `script` 的 `source` 。在 `components` 文件夹目录中我们会放置所有 `components` （个别元件文件夹目录中会用 `index.js` 输出元件，让引入元件更简洁）、 `containers` （负责和 store 互动取得 state），另外还有 `actions` 、 `constants` 、 `reducers` 、 `store` ，其余配置文件则放置于根目录下。

大致上的文件夹目录结构会长这样：



接下来我们参考上一章设定一下开发文档（`.babelrc`、`.eslintrc`、`webpack.config.js`）。这样我们就完成了开发环境的设定可以开始动手实践 `React Redux` 应用程式了！

首先我们先用 Component 之眼感受一下我们应用程式，将它切成一个个 `Component`。在这边我们设计一个主要的 `Main` 包含两个子 Component：`TodoHeader`、`TodoList`。

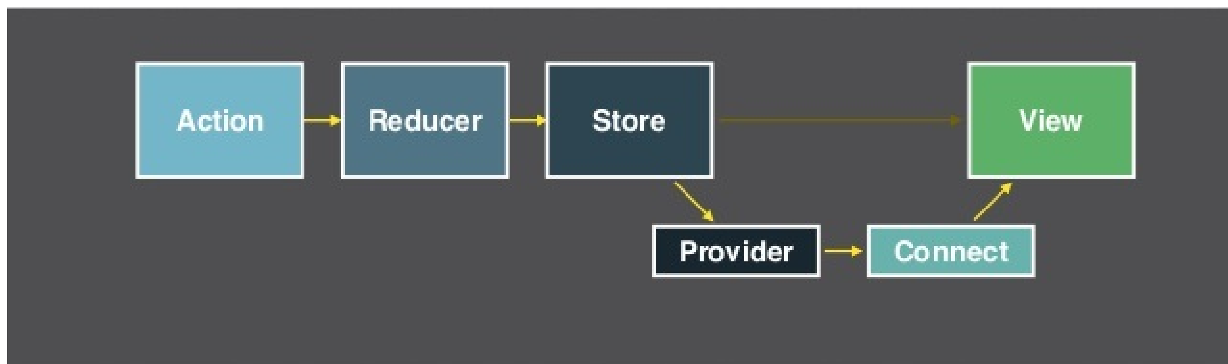


首先设计 HTML Markup：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Redux Todo</title>
</head>
<body>
  <div id="app"></div>
</body>
</html>
```

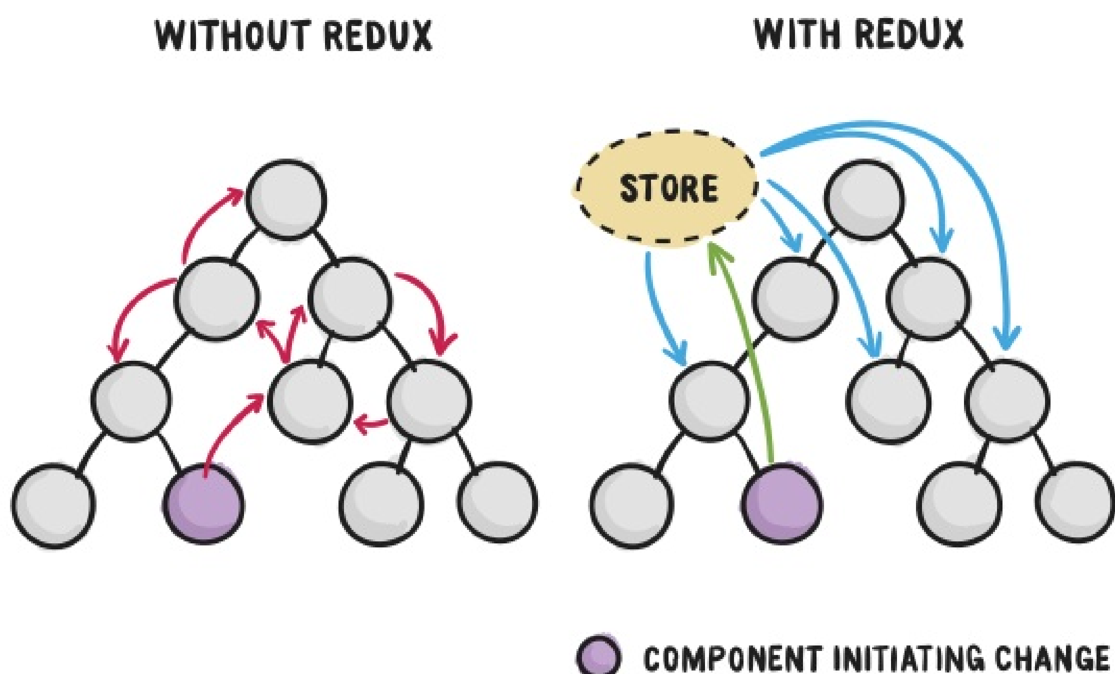
在编写 `src/index.js` 之前，我们先说明整合 `react-redux` 的用法。从以下这张图可以看到 `react-redux` 是 React 和 Redux 间的桥梁，使用 `Provider`、`connect` 去连结 `store` 和 React View。

Data flow with react-redux



Redux using react-redux

事实上，整合了 `react-redux` 后，我们的 React App 就可以解决传统跨 Component 之前传递 state 的问题和困难。只要通过 `Provider` 就可以让每个 React App 中的 `Component` 取用 store 中的 state，非常方便（接下来我们也会更详细说明 Container/Component、 `connect` 的用法）。



以下是 `src/index.js` 完整代码：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import Main from './components/Main';
import store from './store';

ReactDOM.render(
  <Provider store={store}>
    <Main />
  </Provider>,
  document.getElementById('app')
);
```

其中 `src/components/Main/Main.js` 是 Stateless Component，负责所有 View 的进入点。

```
import React from 'react';
import ReactDOM from 'react-dom';
import TodoHeaderContainer from '../../containers/TodoHeaderCont
ainer';
import TodoListContainer from '../../containers/TodoListContaine
r';

const Main = () => (
  <div>
    <TodoHeaderContainer />
    <TodoListContainer />
  </div>
);

export default Main;
```

接下来我们定义一下 `Actions` 的部份，由于是范例 App 所以相对简单，这边只定义一个 todoActions。在这边我们使用了 redux-actions，它可以方便我们使用 Flux Standard Action 格式的 action。以下是 `src/actions/todoActions.js` 完整代码：

```
import { createAction } from 'redux-actions';
import {
  CREATE_TODO,
  DELETE_TODO,
  CHANGE_TEXT,
} from '../constants/actionTypes';

export const createTodo = createAction('CREATE_TODO');
export const deleteTodo = createAction('DELETE_TODO');
export const changeText = createAction('CHANGE_TEXT');
```

我们在 `src/actions/index.js` 将所有 actions 输出

```
export * from './todoActions';
```

另外我们把 constants 放到 `components` 文件夹目录中方便管理，以下是 `src/constants/actionTypes.js` 代码：

```
export const CREATE_TODO = 'CREATE_TODO';
export const DELETE_TODO = 'DELETE_TODO';
export const CHANGE_TEXT = 'CHANGE_TEXT';

/*
或是可以考虑使用 keyMirror，方便产生与 key 相同的常数
import keyMirror from 'fbjs/lib/keyMirror';

export default keyMirror({
    ADD_ITEM: null,
    DELETE_ITEM: null,
    DELETE_ALL: null,
    FILTER_ITEM: null
});
*/
```

设定 Actions 后我们来讨论一下 Reducers 的部份。在讨论 Reducers 之前我们先来设定一下我们的前端的数据结构，在这边我们把所有数据结构（initialState）放到 `src/constants/models.js` 中。这边特别注意的是由于 Redux 中有一个重要

特性是 `State is read-only` ，也就是说更新当 reducers 进到 action 只会回传新的 state 不会更改到原有的 state。因此我们会在整个 Redux App 中使用 `ImmutableJS` 让整个数据流维持在 `Immutable` 的状态，也可以提升程式开发上的性能和避免不可预期的副作用。

以下是 `src/constants/models.js` 完整代码，其设定了 TodoState 的数据结构并使用 `fromJS()` 转成 `Immutable` ：

```
import Immutable from 'immutable';

export const TodoState = Immutable.fromJS({
  'todos': [],
  'todo': {
    id: '',
    text: '',
    updatedAt: '',
    completed: false,
  }
});
```

接下来我们要讨论的是 Reducers 的部份，在 `todoReducers` 中我们会根据接收到的 action 进行 mapping 到对应的处理函数并传入夹带的 `payload` 数据（这边我们使用 redux-actions 来进行 mapping，使用上比传统的 switch 更为简洁）。Reducers 接收到 action 的处理方式为 `(initialState, action) => newState` ，最终会回传一个新的 state，而非更改原来的 state，所以这边我们使用 `ImmutableJS` 。

```
import { handleActions } from 'redux-actions';
import { TodoState } from '../../constants/models';

import {
  CREATE_TODO,
  DELETE_TODO,
  CHANGE_TEXT,
} from '../../constants/actionTypes';

 const todoReducers = handleActions({
  CREATE_TODO: (state) => {
    let todos = state.get('todos').push(state.get('todo'));
    return state.set('todos', todos)
  },
  DELETE_TODO: (state, { payload }) => (
    state.set('todos', state.get('todos').splice(payload.index, 1
))
  ),
  CHANGE_TEXT: (state, { payload }) => (
    state.merge({ 'todo': payload })
  )
}, TodoState);

export default todoReducers;
```

```
import { handleActions } from 'redux-actions';
import UiState from '../../constants/models';

export default handleActions({
  SHOW: (state, { payload }) => (
    state.set('todos', payload.todo)
  ),
}, UiState);
```

虽然 Redux 本身仅会有一个 store，但 redux 本身有提供了 `combineReducers` 可以让我们切割我们 state 方便维护和管理。实上，state 的规划也是一们学问，通常需要不断地实践和工作团队讨论才能找到比较好的方式。不过这边要注意的是我

们改使用了 `redux-immutable` 的 `combineReducers` 这样可以确保我们的 state 维持在 `Immutable` 的状态。

由于 Redux 官方也没有特别明确或严谨的规范。在一般情况我会将 reducers 分为 `data` 和单纯和 UI 有关的 `ui` state。但由于这边是比较简单的例子，我们最终只使用到 `src/reducers/data/todoReducers.js` 。

```
import { combineReducers } from 'redux-immutable';
import ui from './ui/uiReducers';// import routes from './routes
';
import todo from './data/todoReducers';// import routes from './
routes';

const rootReducer = combineReducers({
  todo,
});

export default rootReducer;
```

还记得我们上面说明 React Redux 之前的桥梁时有提到的 store 吗？现在我们要更仔细地去设计 `store` ，我们这边使用到了 redux 其中两个 API：applyMiddleware、createStore。分别可以产生 store 和挂载我们要使用的 middleware（这边我们只使用到 redux-logger 方便我们除错）。注意我们 initialState 也是维持在 `Immutable` 的状态。

```
import { createStore, applyMiddleware } from 'redux';
import createLogger from 'redux-logger';
import Immutable from 'immutable';
import rootReducer from '../reducers';

const initialState = Immutable.Map();

export default createStore(
  rootReducer,
  initialState,
  applyMiddleware(createLogger({ stateTransformer: state => stat
e.toJS() }))
);
```

通过 `src/store/index.js` 输出 configureStore：

```
export { default } from './configureStore';
```

讲解完架构层面的议题，终于我们来到了 View 的部份。加油，距离我们终点也不远了！ 在开始讨论 `Component` 的部份之前我们先来研究一下

react-redux 所提供的 API `connect` 将 props 传给 Component，其用法如下：

`connect([mapStateToProps], [mapDispatchToProps], [mergeProps], [options])`

在我们的范例 App 中我们只会先用到前两个参数，第三个参数会在之后的例子里用到。第一个参数 mapStateToProps 是一个让开发者可以从 store 取出想要 state 并当做 props 往下传的功能，第二个参数则是将 dispatch 行为封装成函数顺著 props 可以方便往下传和调用。

以下是 `src/components/TodoHeader/TodoHeader.js` 的部份：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { connect } from 'react-redux';
import TodoHeader from '../../components/TodoHeader';

// 将欲使用的 actions 引入
import {
  changeText,
  createTodo,
} from '../../actions';

const mapStateToProps = (state) => ({
  // 从 store 取得 todo state
  todo: state.getIn(['todo', 'todo'])
});

const mapDispatchToProps = (dispatch) => ({
  // 当使用者在 input 输入数据值即会触发这个函数，发出 changeText actio
n 并附上使用者输入内容 event.target.value
  onChangeText: (event) => (
    dispatch(changeText({ text: event.target.value }))
```

```
  ),
  // 当使用者按下送出时，发出 createTodo action 并清空 input
  onCreateTodo: () => {
    dispatch(createTodo());
    dispatch(changeText({ text: '' }));
  }
});

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(TodoHeader);

// 开始建设 Component 并使用 connect 进来的 props 并绑定事件（onChange、onClick）。注意我们的 state 因为是使用 `ImmutableJS` 所以要用 `get()` 取值
const TodoHeader = ({
  onChangeText,
  onCreateTodo,
  todo,
}) => (
  <div>
    <h1>TodoHeader</h1>
    <input type="text" value={todo.get('text')} onChange={onChangeText} />
    <button onClick={onCreateTodo}>送出</button>
  </div>
);

export default TodoHeader;
```

以下是 `src/components/TodoList/TodoList.js` 的部份：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { connect } from 'react-redux';
import TodoList from '../../components/TodoList';

import {
```

```
  deleteTodo,
} from '../../actions';

const mapStateToProps = (state) => ({
  todos: state.getIn(['todo', 'todos'])
});

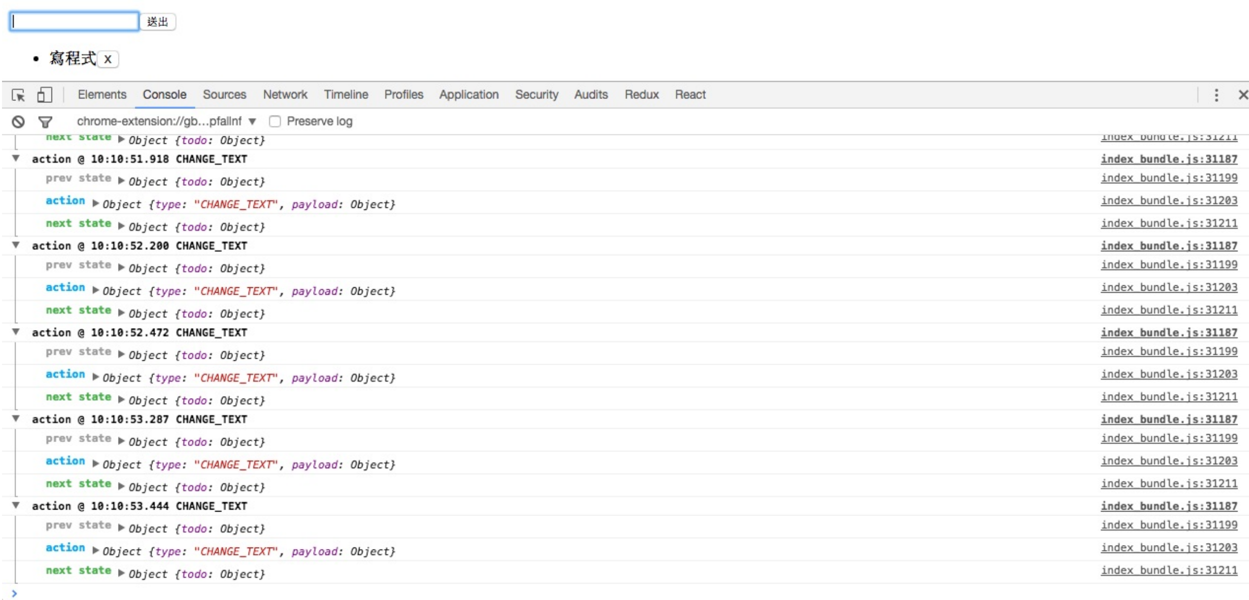
// 由 Component 传进欲删除元素的 index
const mapDispatchToProps = (dispatch) => ({
  onDeleteTodo: (index) => () => (
    dispatch(deleteTodo({ index }))
  )
});

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(TodoList);

// Component 部分值的注意的是 todos state 是通过 map function 去迭代出元素，由于要让 React JSX 可以渲染并保持传入触发 event state 的 immutable，所以需使用 toJS() 转换 component of array。
const TodoList = ({
  todos,
  onDeleteTodo,
}) => (
  <div>
    <ul>
    {
      todos.map((todo, index) => (
        <li key={index}>
          {todo.get('text')}
          <button onClick={onDeleteTodo(index)}>X</button>
        </li>
      )).toJS()
    }
    </ul>
  </div>
);
```

```
export default TodoList;
```

若是一切顺利的话就可以在浏览器上看到自己努力的成果！（因为我们有使用 `redux-logger` 所以打开 console 会看到 action 和 state 的变化情形，但记得在 `production` 环境要拿掉）



总结

以上就是 Redux 实战入门，对于第一次自己动手写 Redux 的朋友可能会需要多练习几次，多体会整个架构。在接下来的章节我们将优化我们的 React Redux TodoApp，让它可以有更清晰好维护的架构。

延伸阅读

1. Redux 官方网站

（image via JonasOhlsson、licdn）

:door: 任意门

| 回首页 | 上一章：Redux 基础概念 | 下一章：Container 与 Presentational Components 入门 |

| 纠错、提问或许愿 |

Ch08 Container 与 Presentational Components

:door: 任意门

| 回首页 |

Container 与 Presentational Components 入门

前言

在聊完了 React 和 Redux 整合后我们来谈谈分离 Presentational 和 Container Component 的概念，若你是第一次听过这个名词，我建议你可以先看看 Redux 作者 Dan AbramovFollow 所写的这篇文章 Presentational and Container Components。

Container 与 Presentational Components 超级比一比

以下先参考 Redux 官网 列出两者相异之处：

1. Presentational Components
    - 用途：怎么看事情（Markup、外观）
    - 是否让 Redux 意识到：否
    - 取得数据方式：从 props 取得
    - 改变数据方式：从 props 去呼叫 callback function
    - 写入方式：手动处理
2. Container Components
    - 用途：怎么做事情（采集数据，更新 State）
    - 是否让 Redux 意识到：是
    - 取得数据方式：订阅 Redux State（store）
    - 改变数据方式：Dispatch Redux Action
    - 写入方式：从 React Redux 产生

    从上面的分析读者可以发现，两者最大的差别在于 `Component` 主要负责单纯的 UI 的渲染，而 `Container` 则负责和 Redux 的 store 沟通，作为 `Redux` 和 `Component` 之间的桥梁。这样的分法可以让程序架构和职责更清楚，所以接下来我们就使用上一章节的 Redux TodoApp 进行改造，改造成 Container 与 Presentational Components 模式。

Container Components

以下是 `src/containers/TodoHeaderContainer/TodoHeaderContainer.js` 的部份：

```
import { connect } from 'react-redux';
import TodoHeader from '../../components/TodoHeader';

// 将欲使用的 actions 引入
import {
  changeText,
  createTodo,
} from '../../actions';

const mapStateToProps = (state) => ({
  // 从 store 取得 todo state
  todo: state.getIn(['todo', 'todo'])
});

const mapDispatchToProps = (dispatch) => ({
  // 当使用者在 input 输入数据值即会触发这个函数，发出 changeText actio
n 并附上使用者输入内容 event.target.value
  onChangeText: (event) => (
    dispatch(changeText({ text: event.target.value }))
  ),
  // 当使用者按下送出时，发出 createTodo action 并清空 input
  onCreateTodo: () => {
    dispatch(createTodo());
    dispatch(changeText({ text: '' }));
  }
});

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(TodoHeader);
```

以下是 `src/containers/TodoListContainer/TodoListContainer.js` 的部份：

```
import { connect } from 'react-redux';
import TodoList from '../../components/TodoList';

import {
  deleteTodo,
} from '../../actions';

const mapStateToProps = (state) => ({
  todos: state.getIn(['todo', 'todos'])
});

const mapDispatchToProps = (dispatch) => ({
  onDeleteTodo: (index) => () => (
    dispatch(deleteTodo({ index }))
  )
});

export default connect(
  mapStateToProps,
  mapDispatchToProps,
)(TodoList);
```

Presentational Components

以下是 `src/components/TodoHeader/TodoHeader.js` 的部份：

```
import React from 'react';
import ReactDOM from 'react-dom';

// 开始建设 Component 并使用 connect 进来的 props 并绑定事件（onChang
e、onClick）。注意我们的 state 因为是使用 `ImmutableJS` 所以要用 `get(
)` 取值

const TodoHeader = ({
  onChangeText,
  onCreateTodo,
  todo,
}) => (
  <div>
    <h1>TodoHeader</h1>
    <input type="text" value={todo.get('text')} onChange={onChan
geText} />
    <button onClick={onCreateTodo}>送出</button>
  </div>
);

export default TodoHeader;
```

以下是 `src/components/TodoList/TodoList.js` 的部份：

```
import React from 'react';
import ReactDOM from 'react-dom';

// Component 部分值的注意的是 todos state 是通过 map function 去迭代出元素，由于要让 React JSX 可以渲染并保持传入触发 event state 的 immutable，所以需使用 toJS() 转换 component of array。
// 由 Component 传进欲删除元素的 index

const TodoList = ({
  todos,
  onDeleteTodo,
}) => (
  <div>
    <ul>
    {
      todos.map((todo, index) => (
        <li key={index}>
          {todo.get('text')}
          <button onClick={onDeleteTodo(index)}>X</button>
        </li>
      )).toJS()
    }
    </ul>
  </div>
);

export default TodoList;
```

总结

That's it！通过区分 Container 与 Presentational Components 可以让程序架构和职责更清楚了！接下来我们将运用我们所学实际开发两个贴近生活的专案，让读者更加熟悉 React 生态系如何应用于实务上。

延伸阅读

1. Presentational and Container Components

2. Redux Usage with React
3. React Higher Order Components in depth
4. React higher order components

:door: 任意门

| 回首页 | 上一章：Redux 实战入门 | 下一章：用 React + Router + Redux + ImmutableJS 写一个 Github 查询应用 |

| 纠错、提问或许愿 |

Ch09 用 React + Router + Redux + ImmutableJS 写一个 Github 查询应用

:door: 任意门

| 回首页 |

用 React + Router + Redux + ImmutableJS 写一个 Github 查找应用

前言

学了一身本领后，本章将带大家完成一个单页式应用程序（Single Page Application），集成 React + Redux + ImmutableJS + React Router 搭配 Github API 制作一个简单的 Github 用户查找应用，实际体验一下开发 React App 的感受。

功能规划

让访客可以使用 Github ID 搜索 Github 用户，展示 Github 用户名、follower、following、avatar_url 并可以返回首页。

使用技术

1. React
2. Redux
3. Redux Thunk
4. React Router
5. ImmutableJS
6. Fetch
7. Material UI
8. Roboto Font from Google Font
9. Github API（https://api.github.com/users/torvalds）

不过要注意的是 Github API 若没有使用 App key 的话可以调用 API 的次数会受限

项目成果截屏

环境安装与设置

1. 安装 Node 和 NPM
2. 安装所需套件

```
$ npm install --save react react-dom redux react-redux react-router immutable redux-immutable redux-actions whatwg-fetch redux-thunk material-ui react-tap-event-plugin
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader ba
bel-preset-es2015 babel-preset-react babel-preset-stage-1 eslint
 eslint-config-airbnb eslint-loader eslint-plugin-import eslint-
plugin-jsx-a11y eslint-plugin-react html-webpack-plugin webpack
webpack-dev-server redux-logger
```

接下来我们先设置一下开发文档。

1. 设置 Babel 的设置档： `.babelrc`

```
{
    "presets": [
      "es2015",
      "react",
     ],
    "plugins": []
}
```

2. 设置 ESLint 的设置档和规则： `.eslintrc`

```
{
  "extends": "airbnb",
  "rules": {
    "react/jsx-filename-extension": [1, { "extensions": [".
js", ".jsx"] }],
  },
  "env" :{
    "browser": true,
  }
}
```

3. 设置 Webpack 设置档： `webpack.config.js`

```
// 让你可以动态插入 bundle 好的 .js 档到 .index.html
const HtmlWebpackPlugin = require('html-webpack-plugin');

const HTMLWebpackPluginConfig = new HtmlWebpackPlugin({
```

```
    template: `${__dirname}/src/index.html`,
    filename: 'index.html',
    inject: 'body',
  });

  // entry 为进入点，output 为进行完 eslint、babel loader 转译后的
文件位置
  module.exports = {
    entry: [
      './src/index.js',
    ],
    output: {
      path: `${__dirname}/dist`,
      filename: 'index_bundle.js',
    },
    module: {
      preLoaders: [
        {
          test: /\.jsx$|\.js$/,
          loader: 'eslint-loader',
          include: `${__dirname}/src`,
          exclude: /bundle\.js$/
        }
      ],
      loaders: [{
        test: /\.js$/,
        exclude: /node_modules/,
        loader: 'babel-loader',
        query: {
          presets: ['es2015', 'react'],
        },
      }],
    },
    // 启动开发测试用 server 设置（不能用在 production）
    devServer: {
      inline: true,
      port: 8008,
    },
    plugins: [HTMLWebpackPluginConfig],
  };
```

太好了！这样我们就完成了开发环境的设置可以开始动手实操 `Github Finder` 应用程序了！

动手实操

1. Setup Mockup

   HTML Markup（`src/index.html`）：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
    <title>GithubFinder</title>
    <link href="https://fonts.googleapis.com/css?family=Roboto:300,400,500" rel="stylesheet">
</head>
<body>
    <div id="app"></div>
</body>
</html>
```

   设置 `webpack.config.js` 的进入点 `src/index.js`：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { browserHistory, Router, Route, IndexRoute } from 'react-router';
import injectTapEventPlugin from 'react-tap-event-plugin';
import MuiThemeProvider from 'material-ui/styles/MuiThemeProvider';
import Main from './components/Main';
import HomePageContainer from './containers/HomePageContainer';
import ResultPageContainer from './containers/ResultPageContainer';
```

```
import store from './store';

// 引入 react-tap-event-plugin 避免 material-ui onTouchTap event 会遇到的问题
// Needed for onTouchTap
// http://stackoverflow.com/a/34015469/988941
injectTapEventPlugin();

// 用 react-redux 的 Provider 包起来将 store 传递下去，让每个 components 都可以访问到 state
// 这边使用 browserHistory 当做 history，并使用 material-ui 的 MuiThemeProvider 包裹整个 components
// 由于这边是简易的 App 我们设计了 Main 为母模版，其有两个子组件 HomePageContainer 和 ResultPageContainer，其中 HomePageContainer 为根位置的子组件
ReactDOM.render(
  <Provider store={store}>
    <MuiThemeProvider>
      <Router history={browserHistory}>
        <Route path="/" component={Main}>
          <IndexRoute component={HomePageContainer} />
          <Route path="/result" component={ResultPageContainer} />
        </Route>
      </Router>
    </MuiThemeProvider>
  </Provider>,
  document.getElementById('app')
);
```

2. Actions

首先先定义 actions 常数：

```
export const SHOW_SPINNER = 'SHOW_SPINNER';
export const HIDE_SPINNER = 'HIDE_SPINNER';
export const GET_GITHUB_INITIATE = 'GET_GITHUB_INITIATE';
export const GET_GITHUB_SUCCESS = 'GET_GITHUB_SUCCESS';
export const GET_GITHUB_FAIL = 'GET_GITHUB_FAIL';
export const CHAGE_USER_ID = 'CHAGE_USER_ID';
```

现在我们来规划我们的 actions 的部份，这个范例我们使用到了 `redux-thunk` 来处理异步的 action（若读者对于新的 Ajax 处理方式 fetch() 不熟悉可以先参考这个文档）。以下是 `src/actions/githubActions.js` 完整代码：

```
// 这边引入了 fetch 的 polyfill，考以让旧的浏览器也可以使用 fetch
import 'whatwg-fetch';
// 引入 actionTypes 常数
import {
  GET_GITHUB_INITIATE,
  GET_GITHUB_SUCCESS,
  GET_GITHUB_FAIL,
  CHAGE_USER_ID,
} from '../constants/actionTypes';

// 引入 uiActions 的 action
import {
  showSpinner,
  hideSpinner,
} from './uiActions';

// 这边是这个范例的重点，要学习我们之前尚未讲解的异步 action 处理方式：不同于一般同步 action 直接发送 action，异步 action 会回传一个带有 dispatch 参数的 function，里面使用了 Ajax（这里使用 fetch()）进行处理
// 一般和 API 交互的流程：INIT（开始请求/秀出 spinner）-> COMPLETE（完成请求/隐藏 spinner）-> ERROR（请求失败）
// 这次我们虽然没有使用 redux-actions 但我们还是维持标准 Flux Standard Action 格式：{ type: '', payload: {} }

export const getGithub = (userId = 'torvalds') => {
```

```
    return (dispatch) => {
      dispatch({ type: GET_GITHUB_INITIATE });
      dispatch(showSpinner());
      fetch('https://api.github.com/users/' + userId)
        .then(function(response) { return response.json() })
        .then(function(json) {
          dispatch({ type: GET_GITHUB_SUCCESS, payload: { dat
a: json } });
          dispatch(hideSpinner());
        })
        .catch(function(response) { dispatch({ type: GET_GITH
UB_FAIL }) });
    }
  }

 // 同步 actions 处理，回传 action 对象
 export const changeUserId = (text) => ({ type: CHAGE_USER_I
D, payload: { userId: text } });
```

以下是 `src/actions/uiActions.js` 负责处理 UI 的行为：

```
import { createAction } from 'redux-actions';
import {
  SHOW_SPINNER,
  HIDE_SPINNER,
} from '../constants/actionTypes';

// 同步 actions 处理，回传 action 对象
export const showSpinner = () => ({ type: SHOW_SPINNER});
export const hideSpinner = () => ({ type: HIDE_SPINNER});
```

透过于 `src/actions/index.js` 将我们 actions 输出

```
export * from './uiActions';
export * from './githubActions';
```

3. Reducers

接下来我们要来设置一下 Reducers 和 models（initialState 格式）的设计，注意我们这个范例都是使用 `ImmutableJS` 。以下是 `src/constants/models.js` ：

```
import Immutable from 'immutable';

export const UiState = Immutable.fromJS({
  spinnerVisible: false,
});

// 我们使用 userId 来暂存用户 ID，data 存放 Ajax 取回的数据
export const GithubState = Immutable.fromJS({
  userId: '',
  data: {},
});
```

以下是 `src/reducers/data/githubReducers.js` ：

```
import { handleActions } from 'redux-actions';
import { GithubState } from '../../constants/models';

import {
  GET_GITHUB_INITIATE,
  GET_GITHUB_SUCCESS,
  GET_GITHUB_FAIL,
  CHAGE_USER_ID,
} from '../../constants/actionTypes';

const githubReducers = handleActions({
  // 当用户按送出按钮，发出 GET_GITHUB_SUCCESS action 时将接收到的数据 merge
  GET_GITHUB_SUCCESS: (state, { payload }) => (
    state.merge({
      data: payload.data,
    })
  ),
  // 当用户输入用户 ID 会发出 CHAGE_USER_ID action 时将接收到的数据 merge
  CHAGE_USER_ID: (state, { payload }) => (
    state.merge({
      'userId':
      payload.userId
    })
  ),
}, GithubState);

export default githubReducers;
```

以下是 `src/reducers/ui/uiReducers.js` :

```
import { handleActions } from 'redux-actions';
import { UiState } from '../../constants/models';

import {
  SHOW_SPINNER,
  HIDE_SPINNER,
} from '../../constants/actionTypes';

// 随着 fetch 结果显示 spinner
const uiReducers = handleActions({
  SHOW_SPINNER: (state) => (
    state.set(
      'spinnerVisible',
      true
    )
  ),
  HIDE_SPINNER: (state) => (
    state.set(
      'spinnerVisible',
      false
    )
  ),
}, UiState);

export default uiReducers;
```

将 reduces 使用 `redux-immutable` 的 `combineReducers` 在一起。以下是 `src/reducers/index.js` ：

```
import { combineReducers } from 'redux-immutable';
import ui from './ui/uiReducers';// import routes from './r
outes';
import github from './data/githubReducers';// import routes
from './routes';

const rootReducer = combineReducers({
  ui,
  github,
});

export default rootReducer;
```

运用 redux 提供的 createStore API 把 `rootReducer` 、 `initialState` 、 `middlewares` 集成后创建出 store。以下是 `src/store/configureSotore.js`

```
import { createStore, applyMiddleware } from 'redux';
import reduxThunk from 'redux-thunk';
import createLogger from 'redux-logger';
import Immutable from 'immutable';
import rootReducer from '../reducers';

const initialState = Immutable.Map();

export default createStore(
  rootReducer,
  initialState,
  applyMiddleware(reduxThunk, createLogger({ stateTransform
er: state => state.toJS() }))
);
```

4. Build Component

终于我们进入了 View 的细节设计，首先我们先针对母模版，也就是每个页面都会出现的 `AppBar` 做设计。以下是 `src/components/Main/Main.js` ：

```
import React from 'react';
// 引入 AppBar
import AppBar from 'material-ui/AppBar';

const Main = (props) => (
  <div>
    <AppBar
      title="Github Finder"
      showMenuIconButton={false}
    />
    <div>
      {props.children}
    </div>
  </div>
);

// 进行 propTypes 验证
Main.propTypes = {
  children: React.PropTypes.object,
};

export default Main;
```

以下是 `src/components/HomePage/HomePage.js` ：

```javascript
import React from 'react';
// 使用 react-router 的 Link 当做超链接，发送 userId 当作 query
import { Link } from 'react-router';
import RaisedButton from 'material-ui/RaisedButton';
import TextField from 'material-ui/TextField';
import IconButton from 'material-ui/IconButton';
import FontIcon from 'material-ui/FontIcon';

const HomePage = ({
  userId,
  onSubmitUserId,
  onChangeUserId,
}) => (
  <div>
    <TextField
      hintText="Please Key in your Github User Id."
      onChange={onChangeUserId}
    />
    <Link to={{
      pathname: '/result',
      query: { userId: userId }
    }}>
      <RaisedButton label="Submit" onClick={onSubmitUserId(
userId)} primary />
    </Link>
  </div>
);

export default HomePage;
```

以下是 `src/components/ResultPage/ResultPage.js`，将 `userId` 当作 `props` 传给 `<GithubBox />`：

```
```javascript
import React from 'react';
import GithubBox from '../../components/GithubBox';

const ResultPage = (props) => (
  <div>
```
```

````
    <GithubBox data={props.data} userId={props.location.query.us
erId} />
  </div>
);

export default ResultPage;
```


以下是 `src/components/GithubBox/GithubBox.js`，负责截取的 Github
数据呈现：

```javascript
import React from 'react';
import { Link } from 'react-router';
// 引入 material-ui 的卡片式组件
import { Card, CardActions, CardHeader, CardMedia, CardTitle, Ca
rdText } from 'material-ui/Card';
// 引入 material-ui 的 RaisedButton
import RaisedButton from 'material-ui/RaisedButton';
// 引入 ActionHome icon
import ActionHome from 'material-ui/svg-icons/action/home';

const GithubBox = (props) => (
  <div>
    <Card>
      <CardHeader
        title={props.data.get('name')}
        subtitle={props.userId}
        avatar={props.data.get('avatar_url')}
      />
      <CardText>
        Followers : {props.data.get('followers')}
      </CardText>
      <CardText>
        Following : {props.data.get('following')}
      </CardText>
      <CardActions>
        <Link to="/">
          <RaisedButton
            label="Back"
````

````
            icon={<ActionHome />}
            secondary={true}
          />
        </Link>
      </CardActions>
    </Card>
  </div>
);

export default GithubBox;
```
````

1. Connect State to Component

   最后，我们要将 Container 和 Component 连接在一起（若忘记了，请先回去复习 Container 与 Presentational Components 入门！）。以下是 `src/containers/HomePage/HomePage.js` ，负责将 userId 和使用到的事件处理方法用 props 传进 component ：
````

```
import { connect } from 'react-redux';
import HomePage from '../../components/HomePage';

import {
  getGithub,
  changeUserId,
} from '../../actions';

export default connect(
  (state) => ({
    userId: state.getIn(['github', 'userId']),
  }),
  (dispatch) => ({
    onChangeUserId: (event) => (
      dispatch(changeUserId(event.target.value))
    ),
    onSubmitUserId: (userId) => () => (
      dispatch(getGithub(userId))
    ),
  }),
  (stateProps, dispatchProps, ownProps) => {
    const { userId } = stateProps;
    const { onSubmitUserId } = dispatchProps;
    return Object.assign({}, stateProps, dispatchProps, own
Props, {
      onSubmitUserId: onSubmitUserId(userId),
    });
  }
)(HomePage);
```

以下是 `src/containers/ResultPage/ResultPage.js` ：

```
import { connect } from 'react-redux';
import ResultPage from '../../components/ResultPage';

export default connect(
  (state) => ({
    data: state.getIn(['github', 'data'])
  }),
  (dispatch) => ({})
)(ResultPage);
```

2. That's it

若一切顺利的话，这时候你可以在终端机下 `$ npm start` 指令，然后在 `http://localhost:8008` 就可以看到你的努力成果啰！



总结

本章带领读者们从零开始集成 React + Redux + ImmutableJS + React Router 搭配 Github API 制作一个简单的 Github 用户查找应用。下一章我们将挑战高端应用，学习 Server Side Rendering 方面的知识，并用 React + Redux + Node（Isomorphic）开发一个食谱分享网站。

延伸阅读

1. Tutorial: build a weather app with React
2. OpenWeatherMap
3. Weather Icons
4. Weather API Icons
5. Material UI
6. 【翻译】这个API很“迷人”——(新的Fetch API)
7. Redux: trigger async data fetch on React view event
8. Github API
9. 传统 Ajax 已死，Fetch 永生

:door: 任意门

| 回首页 | 上一章：Container 与 Presentational Components 入门 | 下一章：React Redux Sever Rendering（Isomorphic JavaScript）入门 |

| 纠错、提问或许愿 |

Ch10 实战教学：用 React + Redux + Node（Isomorphic JavaScript）开发食谱分享网站

1. React Redux Sever Rendering（Isomorphic JavaScript）入门
2. 用 React + Redux + Node（Isomorphic JavaScript）开发一个食谱分享网站

:door: 任意门

| 回首页 |

React Redux Sever Rendering（Isomorphic JavaScript）入门



前言

由于可能有些读者没听过 Isomorphic JavaScript 。因此在进到开发 React Redux Sever Rendering 应用程序的主题之前我们先来聊聊 Isomorphic JavaScript 这个议题。

根据 Isomorphic JavaScript 这个网站的说明：

> Isomorphic JavaScript Isomorphic JavaScript apps are JavaScript applications that can run both client-side and server-side. The backend and frontend share the same code.

Isomorphic JavaScript 系指浏览器端和服务器端共用 JavaScript 的代码。

另外，除了 Isomorphic JavaScript 外，读者或许也有听过 Universal JavaScript 这个用词。那什么是 Universal JavaScript 呢？它和 Isomorphic JavaScript 是指一样的意思吗？针对这个议题网络上有些开发者提出了自己的观点： Universal JavaScript、Isomorphism vs Universal JavaScript。其中 Isomorphism vs Universal JavaScript 这篇文章的作者 Gert Hengeveld 指出 `Isomorphic JavaScript` 主要是指前后端共用 JavaScript 的开发方式，而 `Universal JavaScript` 是指 JavaScript 代码可以在不同环境下运行，这当然包含浏览器端和

服务器端，甚至其他环境。也就是说 `Universal JavaScript` 在意义上可以涵盖的比 `Isomorphic JavaScript` 更广泛一些，然而在 Github 或是许多技术讨论上通常会把两者视为同一件事情，这部份也请读者留意。

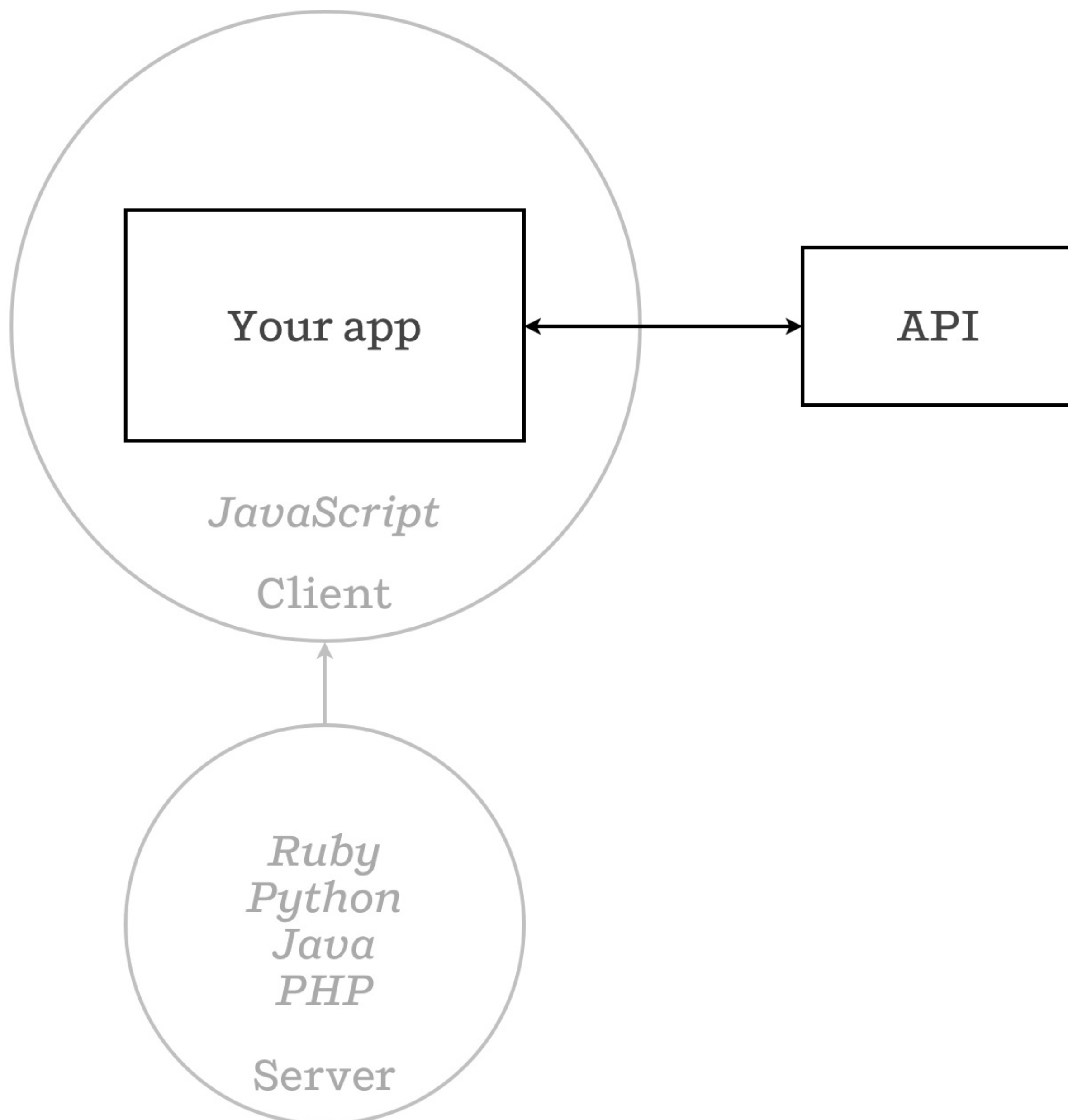
Isomorphic JavaScript 的好处

在开始真正撰写 Isomorphic JavaScript 前我们在进一步探讨使用 Isomorphic JavaScript 有哪些好处？在谈好处之前，我们先看看最早 Web 开发是如何处理页面渲染和 state 管理，还有遇到哪些挑战。

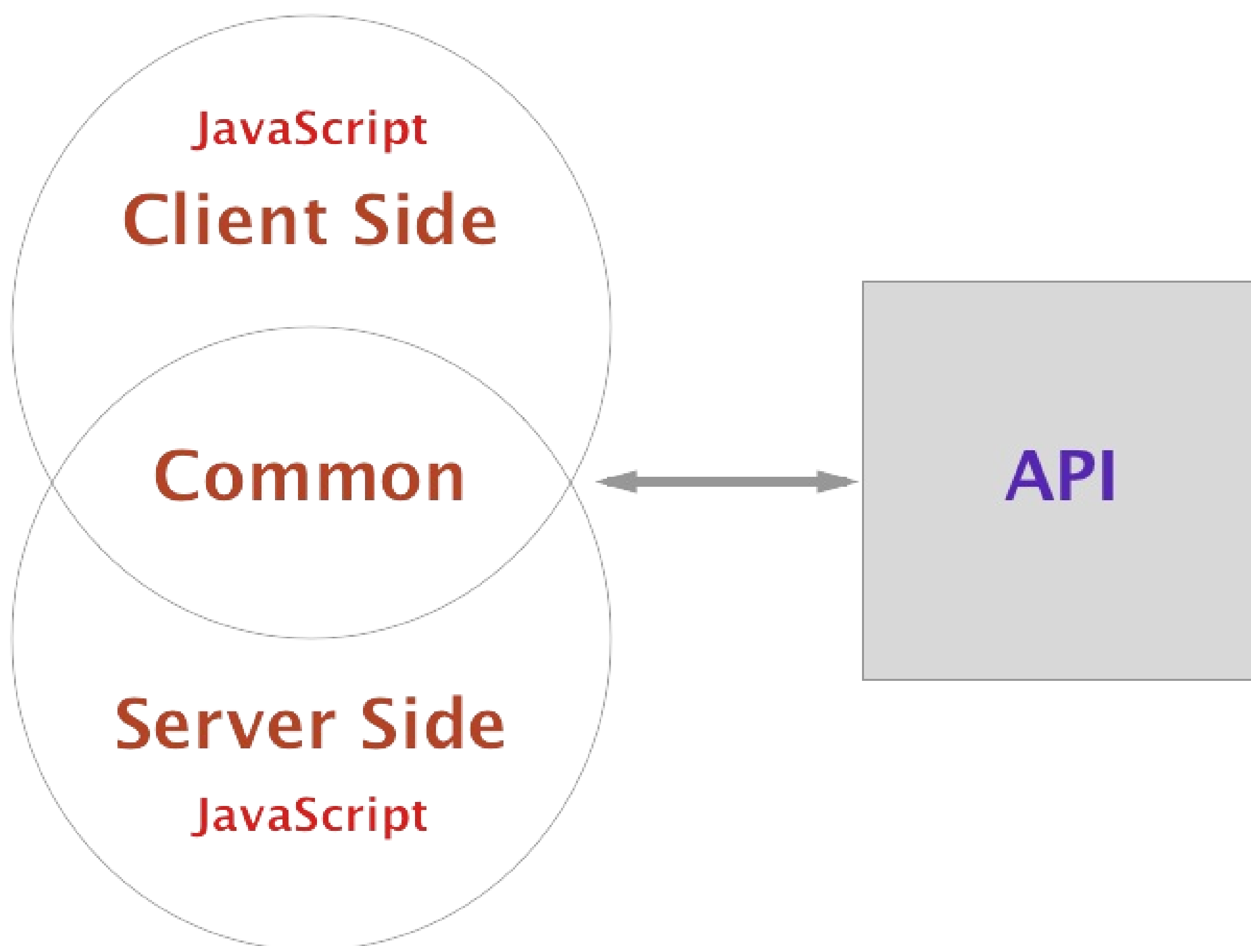
最早的时候我们谈论 Web 很单纯，都是由 Server 端进行模版的处理，你可以想成 template 是一个函数，我们发送数据进去，template 最后产生一张 HTML 给浏览器显示。例如：Node 使用的（EJS、Jade）、Python/Django 的 Template 或替代方案 Jinja、PHP 的 Smarty、Laravel 使用的 Blade，甚至是 Ruby on Rails 用的 ERB。都是由后端去 render 所有数据和页面，前端处理相对单纯。

然而随着前端工程的软件工程化和用户体验的要求，开始出现各式前端框架的百花齐放，例如：Backbone.js、Ember.js 和 Angular.js 等前端 MVC (Model-View-Controller) 或 MVVM (Model-View-ViewModel) 框架，将页面于前端渲染的不刷页单页式应用程序（Single Page App）也因此开始流行。

后端除了提供初始的 HTML 外，还提供 API Server 让前端框架可以取得数据用于前端 template。复杂的逻辑由 ViewModel/Presenter 来处理，前端 template 只处理简单的是否显示或是元素迭代的状况，如下图所示：



然而前端渲染 template 虽然有它的好处但也遇到一些问题包括性能、SEO 等议题。此时我们就开始思考 Isomorphic JavaScript 的可能性：为什么我们不能前后端都使用 JavaScript 甚至是 React？



事实上，React 的优势就在于它可以很优雅地实现 Server Side Rendering 达到 Isomorphic JavaScript 的效果。在 `react-dom/server` 中有两个方法 `renderToString` 和 `renderToStaticMarkup` 可以在 server 端渲染你的 components。其主要都是将 React Component 在 Server 端转成 DOM String，也可以将 props 往下传，然而事件处理会失效，要到 client-side 的 React 接收到后才会把它加上去（但要注意 server-side 和 client-side 的 checksum 要一致不然会出现错误），这样一来可以提高渲染速度和 SEO 效果。 `renderToString` 和 `renderToStaticMarkup` 最大的差异在于 `renderToStaticMarkup` 会少加一些 React 内部使用的 DOM 属性，例如： `data-react-id` ，因此可以节省一些资源。

使用 `renderToString` 进行 Server 端渲染：

```
import ReactDOMServer from 'react-dom/server';

ReactDOMServer.renderToString(<HelloButton name="Mark" />);
```

渲染出来的效果：

```
<button data-reactid=".7" data-react-checksum="762752829">
  Hello, Mark
</button>
```

总的来说使用 Isomorphic JavaScript 会有以下的好处：

1. 有助于 SEO
2. Rendering 速度较快，性能较佳
3. 放弃蹩脚的 Template 语法拥抱 Component 组件化思考，便于维护
4. 尽量前后端共用代码节省开发时间

不过要注意的是如果有使用 Redux 在 Server Side Rendering 中，其流程相对复杂，不过大致流程如下：由后端预先加载需要的 initialState，由于 Server 渲染必须全部都转成 string，所以先将 state 先 dehydration（脱水），等到 client 端再 rehydration（覆水），重建 store 往下传到前端的 React Component。

而要把数据从服务器端传递到客户端，我们需要：

1. 把取得初始 state 当做参数并对每个请求创建一个全新的 Redux store 实体
2. 选择性地 dispatch 一些 action
3. 把 state 从 store 取出来
4. 把 state 一起传到客户端

接下来我们就开始动手实作一个简单的 React Server Side Rendering Counter 应用程序。

项目成果截屏

Clicked: 46 times + -

环境安装与设置

1. 安装 Node 和 NPM
2. 安装所需套件

```
$ npm install --save react react-dom redux react-redux react
-router immutable redux-immutable redux-actions redux-thunk
babel-polyfill babel-register body-parser express morgan qs
```

```
$ npm install --save-dev babel-core babel-eslint babel-loade
r babel-preset-es2015 babel-preset-react babel-preset-stage-
1 eslint eslint-config-airbnb eslint-loader eslint-plugin-im
port eslint-plugin-jsx-a11y eslint-plugin-react html-webpack
-plugin webpack webpack-dev-server redux-logger
```

接下来我们先设置一下开发文档。

1. 设置 Babel 的设置档： `.babelrc`

```
{
 "presets": [
   "es2015",
   "react",
 ],
 "plugins": []
}
```

2. 设置 ESLint 的设置档和规则： `.eslintrc`

```
{
 "extends": "airbnb",
 "rules": {
   "react/jsx-filename-extension": [1, { "extensions": [".js"
, ".jsx"] }],
 },
 "env" :{
   "browser": true,
 }
}
```

3. 设置 Webpack 设置档： `webpack.config.js`

```
// 让你可以动态插入 bundle 好的 .js 档到 .index.html
const HtmlWebpackPlugin = require('html-webpack-plugin');

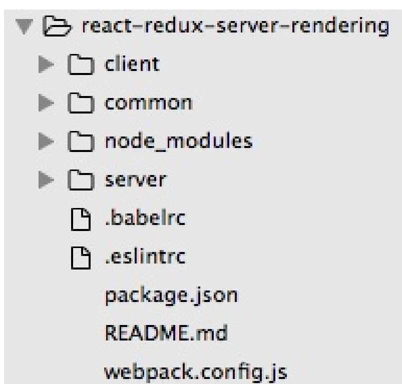
const HTMLWebpackPluginConfig = new HtmlWebpackPlugin({
 template: `${__dirname}/src/index.html`,
 filename: 'index.html',
 inject: 'body',
});

// entry 为进入点，output 为进行完 eslint、babel loader 转译后的
文件位置
module.exports = {
 entry: [
   './src/index.js',
```

```
  ],
  output: {
    path: `${__dirname}/dist`,
    filename: 'index_bundle.js',
  },
  module: {
    preLoaders: [
      {
        test: /\.jsx$|\.js$/,
        loader: 'eslint-loader',
        include: `${__dirname}/src`,
        exclude: /bundle\.js$/
      }
    ],
    loaders: [{
      test: /\.js$/,
      exclude: /node_modules/,
      loader: 'babel-loader',
      query: {
        presets: ['es2015', 'react'],
      },
    }],
  },
  // 启动开发测试用 server 设置（不能用在 production）
  devServer: {
    inline: true,
    port: 8008,
  },
  plugins: [HTMLWebpackPluginConfig],
};
```

太好了！这样我们就完成了开发环境的设置可以开始动手实作 `React Server Side Rendering Counter` 应用程序了！

先看一下我们整个专案的数据结构，我们把整个专案分成三个主要的文件夹（`client` 、`server` ，还有共用代码的 `common` ）：



动手实作

首先，我们先定义了 `client` 的 `index.js` ：

```
// 引用 babel-polyfill 避免浏览器不支持部分 ES6 用法
import 'babel-polyfill';
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import CounterContainer from '../common/containers/CounterContai
ner';
import configureStore from '../common/store/configureStore'
import { fromJS } from 'immutable';

// 从 server 取得传进来的 initialState。由于从字串转回对象，又称为 rehy
dration（覆水）
const initialState = window.__PRELOADED_STATE__;

// 由于我们使用 ImmutableJS，所以需要把在 server-side dehydration（脱
水）又在前端 rehydration（覆水）的 initialState 转成 ImmutableJS 数据
型态，并传进 configureStore 创建 store
const store = configureStore(fromJS(initialState));

// 接下来就跟一般的 React App 一样，把 store 透过 Provider 往下传到 Co
mponent 中
ReactDOM.render(
  <Provider store={store}>
    <CounterContainer />
  </Provider>,
  document.getElementById('app')
);
```

由于 Node 端要到新版对于 ES6 支持较好，所以先用 `babel-register` 在 `src/server/index.js` 去即时转译 `server.js` ，但目前不建议在 `production` 环境使用。

```
// use babel-register to precompile ES6 syntax
require('babel-register');
require('./server');
```

接着是我们 `server` 端，也是这个范例最重要的一个部分。首先我们用 `express` 创建了一个 port 为 3000 的 server，并使用 webpack 去运行 `client` 的代码。这个范例中我们使用了 `handleRender` 当 request 进来时（直接拜访页面或刷新）就会运行 fetchCounter() 进行处理：

```
import Express from 'express';
import qs from 'qs';

import webpack from 'webpack';
import webpackDevMiddleware from 'webpack-dev-middleware';
import webpackHotMiddleware from 'webpack-hot-middleware';
import webpackConfig from '../webpack.config';

import React from 'react';
import { renderToString } from 'react-dom/server';
import { Provider } from 'react-redux';
import { fromJS } from 'immutable';

import configureStore from '../common/store/configureStore';
import CounterContainer from '../common/containers/CounterContai
ner';

import { fetchCounter } from '../common/api/counter';

const app = new Express();
const port = 3000;

function handleRender(req, res) {
  // 模仿实际异步 api 处理情形
  fetchCounter(apiResult => {
  // 读取 api 提供的数据（这边我们 api 是用 setTimeout 进行模仿异步状况
），若网址参数有值择取值，若无则使用 api 提供的随机值，若都没有则取 0
    const params = qs.parse(req.query);
    const counter = parseInt(params.counter, 10) || apiResult ||
0;
    // 将 initialState 转成 immutable 和符合 state 设计的格式
    const initialState = fromJS({
      counterReducers: {
        count: counter,
```

```
      }
    });
    // 创建一个 redux store
    const store = configureStore(initialState);
    // 使用 renderToString 将 component 转为 string
    const html = renderToString(
      <Provider store={store}>
        <CounterContainer />
      </Provider>
    );
    // 从创建的 redux store 中取得 initialState
    const finalState = store.getState();
    // 将 HTML 和 initialState 传到 client-side
    res.send(renderFullPage(html, finalState));
  })
}

// HTML Markup，同时也把 preloadedState 转成字串（stringify）传到 cl
ient-side，又称为 dehydration（脱水）
function renderFullPage(html, preloadedState) {
  return `
    <!doctype html>
    <html>
      <head>
        <title>Redux Universal Example</title>
      </head>
      <body>
        <div id="app">${html}</div>
        <script>


        </script>
        <script src="/static/bundle.js"></script>
      </body>
    </html>
    `
}

// 使用 middleware 于 webpack 去进行 hot module reloading
const compiler = webpack(webpackConfig);
```

```
app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPat
h: webpackConfig.output.publicPath }));
app.use(webpackHotMiddleware(compiler));
// 每次 server 接到 request 都会调用 handleRender
app.use(handleRender);

// 监听 server 状况
app.listen(port, (error) => {
  if (error) {
    console.error(error)
  } else {
    console.info(`==>   Listening on port ${port}. Open up http:
//localhost:${port}/ in your browser.`)
  }
});
```

处理完 Server 的部份接下来我们来处理 actions 的部份，在这个范例中 actions 相对简单，主要就是添加和减少两个行为，以下为 `src/actions/counterActions.js` ：

```
import { createAction } from 'redux-actions';
import {
  INCREMENT_COUNT,
  DECREMENT_COUNT,
} from '../constants/actionTypes';

export const incrementCount = createAction(INCREMENT_COUNT);
export const decrementCount = createAction(DECREMENT_COUNT);
```

以下为输出常数 `src/constants/actionTypes.js` ：

```
export const INCREMENT_COUNT = 'INCREMENT_COUNT';
export const DECREMENT_COUNT = 'DECREMENT_COUNT';
```

在这个范例中我们使用 `setTimeout()` 来仿眞异步的产生数据让 server 端在每次接收 request 时读取随机产生的值。实务上，我们会开 API 让 Server 读取初始要导入的 initialState。

```
function getRandomInt(min, max) {
  return Math.floor(Math.random() * (max - min)) + min
}

export function fetchCounter(callback) {
  setTimeout(() => {
    callback(getRandomInt(1, 100))
  }, 500)
}
```

谈完 actions 我们来看我们的 reducers，在这个范例中 reducers 也是相对简单的，主要就是针对添加和减少两个行为去 set 值，以下是 `src/reducers/counterReducers.js`：

```
import { fromJS } from 'immutable';
import { handleActions } from 'redux-actions';
import { CounterState } from '../constants/models';

import {
  INCREMENT_COUNT,
  DECREMENT_COUNT,
} from '../constants/actionTypes';

const counterReducers = handleActions({
  INCREMENT_COUNT: (state) => (
    state.set(
      'count',
      state.get('count') + 1
    )
  ),
  DECREMENT_COUNT: (state) => (
    state.set(
      'count',
      state.get('count') - 1
    )
  ),
}, CounterState);

export default counterReducers;
```

准备好了 `rootReducer` 就可以使用 `createStore` 来创建我们 store，值得注意的是由于 `configureStore` 需要被 client-side 和 server-side 使用，所以把它输出成 function 方便传入 initialState 使用。以下是 `src/store/configureStore.js` ：

```
import { createStore, applyMiddleware } from 'redux';
import thunk from 'redux-thunk';
import createLogger from 'redux-logger';
import rootReducer from '../reducers';

export default function configureStore(preloadedState) {
  const store = createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(createLogger({ stateTransformer: state => st
ate.toJS() }), thunk)
  )
  return store
}
```

最后来到了 `components` 和 `containers` 的时间，这次我们的 Component 主要有两个按钮让用户可以添加和减少数字并显示目前数字。以下是 `src/components/Counter/Counter.js` ：

```
import React, { Component, PropTypes } from 'react'

const Counter = ({
  count,
  onIncrement,
  onDecrement,
}) => (
  <p>
    Clicked: {count} times
    {' '}
    <button onClick={onIncrement}>
      +
    </button>
    {' '}
    <button onClick={onDecrement}>
      -
    </button>
    {' '}
  </p>
);

// 注意要检查 propTypes 和给定默认值
Counter.propTypes = {
  count: PropTypes.number.isRequired,
  onIncrement: PropTypes.func.isRequired,
  onDecrement: PropTypes.func.isRequired
}

Counter.defaultProps = {
  count: 0,
  onIncrement: () => {},
  onDecrement: () => {}
}

export default Counter;
```

最后把取出的 `count` 和事件处理方法用 connect 传到 `Counter` 就大功告成了！以下是 `src/containers/CounterContainer/CounterContainer.js` ：

```
import 'babel-polyfill';
import { connect } from 'react-redux';
import Counter from '../../components/Counter';

import {
  incrementCount,
  decrementCount,
} from '../../actions';

export default connect(
  (state) => ({
    count: state.get('counterReducers').get('count'),
  }),
  (dispatch) => ({
    onIncrement: () => (
      dispatch(incrementCount())
    ),
    onDecrement: () => (
      dispatch(decrementCount())
    ),
  })
)(Counter);
```

若一切顺利，在终端机打上 `$ npm start` ，你将可以在浏览器的 `http://localhost:3000` 看到自己的成果！

Clicked: 46 times + -

总结

本章阐述了 Web 页面浏览的进程和 Isomorphic JavaScript 的优势，并介绍了如何使用 React Redux 进行 Server Side Rendering 的应用编程。下一个章节我们将集成后端数据库，运用 React + Redux + Node（Isomorphic）开发一个简单的食谱分享网站。

延伸阅读

1. DavidWells/isomorphic-react-example
2. RickWong/react-isomorphic-starterkit
3. Server-rendered React components in Rails
4. Our First Node.js App: Backbone on the Client and Server
5. Going Isomorphic with React
6. A service for server-side rendering your JavaScript views
7. Isomorphic JavaScript: The Future of Web Apps
8. React Router Server Rendering

（image via airbnb）

:door: 任意门

| 回首页 | 上一章：用 React + Router + Redux + ImmutableJS 写一个 Github 查找应用 | 下一章：用 React + Redux + Node（Isomorphic JavaScript）开发食谱分享网站 |

| 纠错、提问或许愿 |

用 React + Redux + Node（Isomorphic JavaScript）开发食谱分享网站

前言

如果你是从一开始跟着我们踏出 React 旅程的读者眞的恭喜你，也谢谢你一路跟着我们的学习脚步，对一个初学者来说这一段路并不容易。本章是扣除附录外我们最后一个正式章节的范例，也是规模最大的一个，在这个章节中我们要整合过去所学和添加一些知识开发一个可以登录会员并分享食谱的社群网站， Let's GO！

需求规划

让使用者可以登录会员并分享食谱的社群网站

功能规划

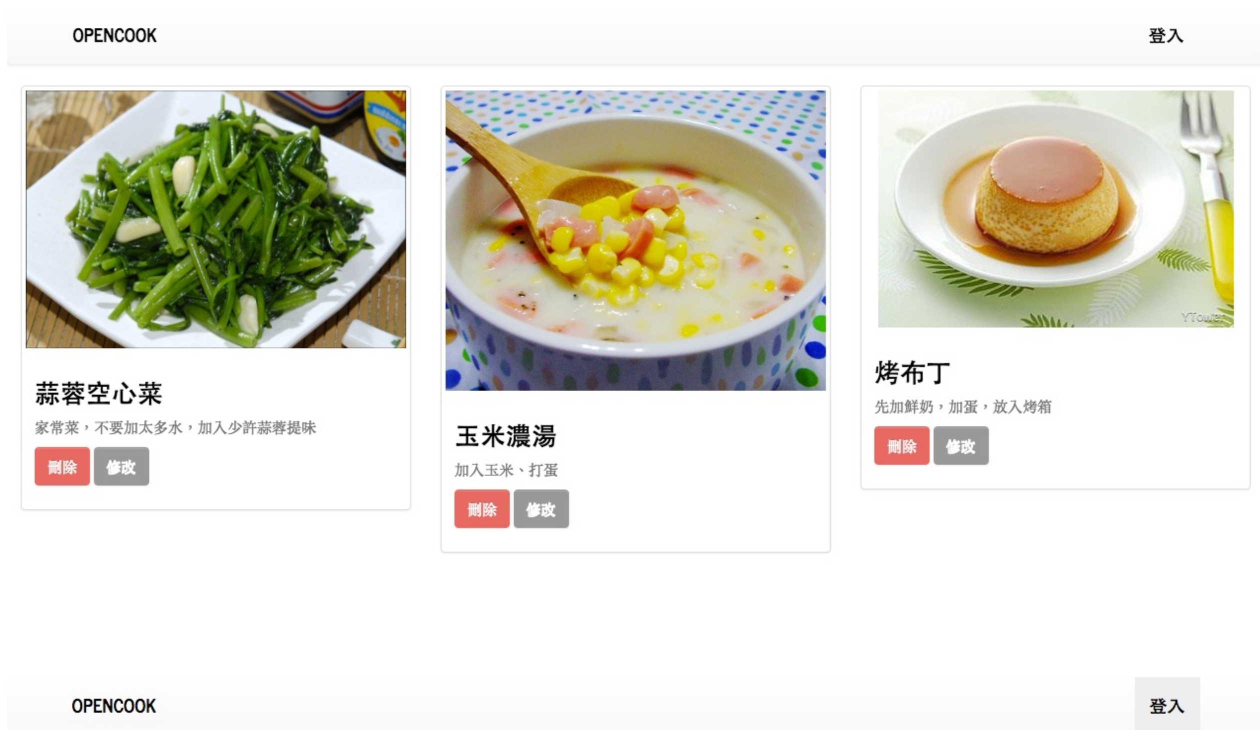
1. React Router / Redux / Immutable / Server Render / Async API
2. 使用者登录/登出（JSON Web Token）
3. CRUD 表单资料处理
4. 资料库串接(ORM/MongoDB)

使用技术

1. React
2. Redux(redux-actions/redux-promise/redux-immutable)
3. React Router
4. ImmutableJS
5. Node MongoDB ORM(Mongoose)
6. JSON Web Token
7. React Bootstrap
8. Axios(Promise)
9. Webpack

10. UUID

项目成果截图



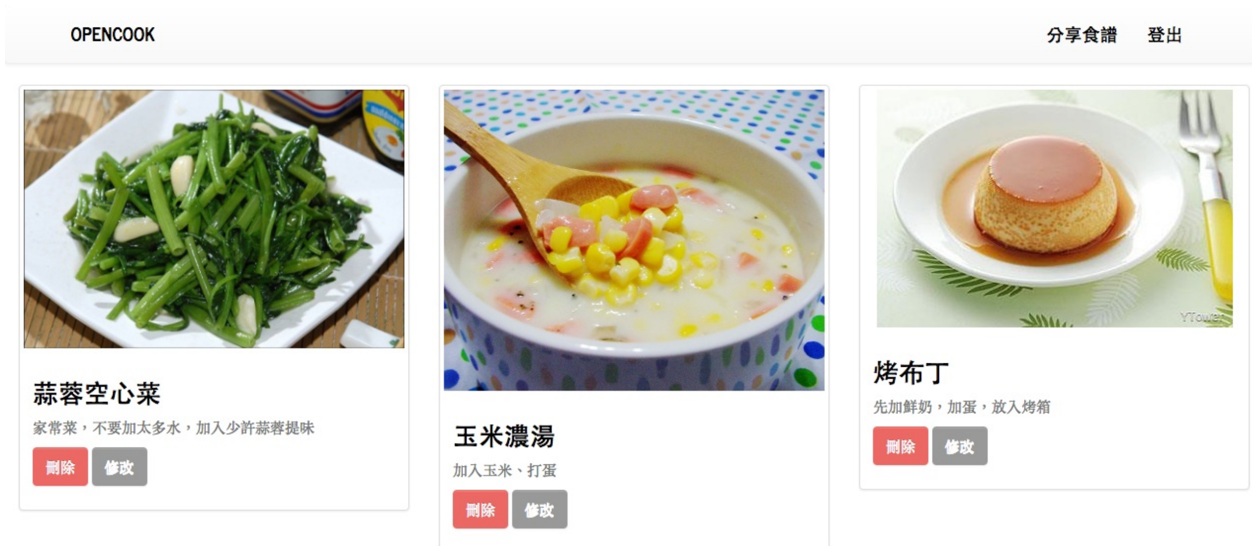
請輸入您的 Email

Enter Email

請輸入您的密碼

Enter Password

提交送出



OPENCOOK 分享食譜 登出

請輸入食譜名稱

Enter text

請輸入食譜說明

textarea

請輸入食譜圖片網址

Enter text

提交送出

环境安装与设定

1. 安装 Node 和 NPM
2. 安装所需套件

```
$ npm install --save react react-dom redux react-redux react-rou
ter immutable redux-immutable redux-actions redux-promise bcrypt
 body-parser cookie-parser debug express immutable jsonwebtoken
mongoose morgan passport passport-local react-router-bootstrap a
xios serve-favicon validator uuid
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader ba
bel-preset-es2015 babel-preset-react babel-preset-stage-1 eslint
 eslint-config-airbnb eslint-loader eslint-plugin-import eslint-
plugin-jsx-a11y eslint-plugin-react html-webpack-plugin webpack
webpack-dev-server redux-logger
```

接下来我们先设定一下开发文档。

1. 设定 Babel 的设定档： `.babelrc`

```
{
 "presets": [
   "es2015",
   "react",
 ],
 "plugins": []
}
```

2. 设定 ESLint 的设定档和规则： `.eslintrc`

```
{
 "extends": "airbnb",
 "rules": {
   "react/jsx-filename-extension": [1, { "extensions": [".js"
, ".jsx"] }],
 },
 "env" :{
   "browser": true,
 }
}
```

3. 设定 Webpack 设定档： `webpack.config.js` ：

```
import webpack from 'webpack';

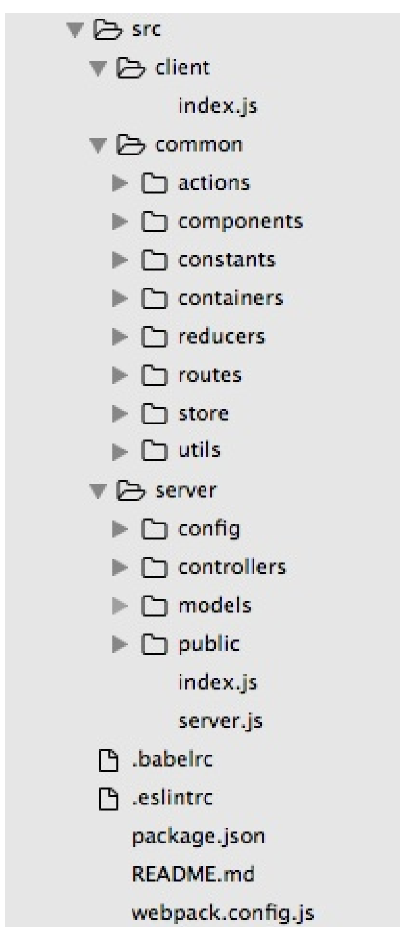
module.exports = {
 entry: [
   './src/client/index.js',
 ],
 output: {
   path: `${__dirname}/dist`,
   filename: 'bundle.js',
   publicPath: '/static/'
 },
 module: {
   preLoaders: [
     {
       test: /\.jsx$|\.js$/,
       loader: 'eslint-loader',
       include: `${__dirname}/app`,
       exclude: /bundle\.js$/,
     },
   ],
   // 使用 Hot Module Replacement 外挂
   plugins: [
     new webpack.optimize.OccurrenceOrderPlugin(),
     new webpack.HotModuleReplacementPlugin()
   ],
   loaders: [{
     test: /\.js$/,
     exclude: /node_modules/,
     loader: 'babel-loader',
     query: {
       presets: ['es2015', 'react'],
     },
   }],
 },
};
```

4. 设定 `src/server/config/index.js` ：

```
export default ({
  "secret": "ilovecooking",
  "database": "mongodb://localhost/open_cook"
});
```

太好了！这样我们就完成了开发环境的设定可以开始动手实作我们的食谱分享社群应用程序了！

同时我们也初步设计我们文件夹结构，主要我们将文件夹分为 `client` 、 `common` 、 `server` ：



动手实操

首先我们先进行 `src/client/index.js` 的设计：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { Provider } from 'react-redux';
import { browserHistory, Router } from 'react-router';
import { fromJS } from 'immutable';
// 我们的 routing 放置在 common 文件夹中的 routes
import routes from '../common/routes';
import configureStore from '../common/store/configureStore';
import { checkAuth } from '../common/actions';

// 将 server side 传过来的 initialState 给 rehydration（覆水）
const initialState = window.__PRELOADED_STATE__;

// 将 initialState 传给 configureStore 函数创建出 store 并传给 Provi
der
const store = configureStore(fromJS(initialState));
ReactDOM.render(
  <Provider store={store}>
    <Router history={browserHistory} routes={routes} />
  </Provider>,
  document.getElementById('app')
);
```

由于 Node 端要到新版对于 ES6 支持较好，所以先用 `babel-register` 在 `src/server/index.js` 去即时转译 `server.js` ，但不建议在 `production` 环境使用。

```
// use babel-register to precompile ES6
require('babel-register');
require('./server');
```

```
// 引入 Express、mongoose（MongoDB ORM）以及相关 server 上使用的套件
/* Server Packages */
import Express from 'express';
import bodyParser from 'body-parser';
import cookieParser from 'cookie-parser';
import morgan from 'morgan';
```

```
import mongoose from 'mongoose';
import config from './config';
// 引入后端 model 透过 model 和数据库互动
import User from './models/user';
import Recipe from './models/recipe';

// 引入 webpackDevMiddleware 当做前端 server middleware
/* Client Packages */
import webpack from 'webpack';
import React from 'react';
import webpackDevMiddleware from 'webpack-dev-middleware';
import webpackHotMiddleware from 'webpack-hot-middleware';
import { RouterContext, match } from 'react-router';
import { renderToString } from 'react-dom/server';
import { Provider } from 'react-redux';
import Immutable, { fromJS } from 'immutable';
/* Common Packages */
import webpackConfig from '../../webpack.config';
import routes from '../common/routes';
import configureStore from '../common/store/configureStore';
import fetchComponentData from '../common/utils/fetchComponentDa
ta';
import apiRoutes from './controllers/api.js';
/* config */
// 初始化 Express server
const app = new Express();
const port = process.env.PORT || 3000;
// 连接到数据库，相关设定档案放在 config.database
mongoose.connect(config.database); // connect to database
app.set('env', 'production');
// 设定静态档案位置
app.use('/static', Express.static(__dirname + '/public'));
app.use(cookieParser());
// use body parser so we can get info from POST and/or URL param
eters
app.use(bodyParser.urlencoded({ extended: false })); // only can
 deal with key/value
app.use(bodyParser.json());
// use morgan to log requests to the console
app.use(morgan('dev'));
```

```
// 负责每次接受到 request 的处理函数，判断该如何处理和取得 initialState
整理后结合服务器渲染页面传往前端
const handleRender = (req, res) => {
  // Query our mock API asynchronously
  match({ routes, location: req.url }, (error, redirectLocation,
 renderProps) => {
    if (error) {
      res.status(500).send(error.message);
    } else if (redirectLocation) {
      res.redirect(302, redirectLocation.pathname + redirectLoca
tion.search);
    } else if (renderProps == null) {
      res.status(404).send('Not found');
    }
    fetchComponentData(req.cookies.token).then((response) => {
      let isAuthorized = false;
      if (response[1].data.success === true) {
         isAuthorized = true;
      } else {
        isAuthorized = false;
      }
      const initialState = fromJS({
        recipe: {
          recipes: response[0].data,
          recipe: {
            id: '',
            name: '',
            description: '',
            imagePath: '',
          }
        },
        user: {
          isAuthorized: isAuthorized,
          isEdit: false,
        }
      });
      // server side 渲染页面
      // Create a new Redux store instance
      const store = configureStore(initialState);
```

```
      const initView = renderToString(
        <Provider store={store}>
          <RouterContext {...renderProps} />
        </Provider>
      );
      let state = store.getState();
      let page = renderFullPage(initView, state);
      return res.status(200).send(page);
    })
    .catch(err => res.end(err.message));
  })
}

// 基础页面 HTML 设计
const renderFullPage = (html, preloadedState) => (`
    <!doctype html>
    <html>
      <head>
        <title>OpenCook 分享料理的美好时光</title>
        <!-- Latest compiled and minified CSS -->
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/b
ootstrap/latest/css/bootstrap.min.css">
        <!-- Optional theme -->
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/b
ootstrap/latest/css/bootstrap-theme.min.css">
        <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/b
ootswatch/3.3.7/journal/bootstrap.min.css">
      <body>
        <div id="app">${html}</div>
        <script>
          window.__PRELOADED_STATE__ = ${JSON.stringify(preloade
dState).replace(/</g, '\\x3c')}
        </script>
        <script src="/static/bundle.js"></script>
      </body>
    </html>`
);

// 设定 hot reload middleware
const compiler = webpack(webpackConfig);
```

```
app.use(webpackDevMiddleware(compiler, { noInfo: true, publicPat
h: webpackConfig.output.publicPath }));
app.use(webpackHotMiddleware(compiler));

// 设计 API prefix，并使用 controller 中的 apiRoutes 进行处理
app.use('/api', apiRoutes);
// 使用服务器端 handleRender
app.use(handleRender);
app.listen(port, (error) => {
  if (error) {
    console.error(error)
  } else {
    console.info(`==>   Listening on port ${port}. Open up http:
//localhost:${port}/ in your browser.`)
  }
});
```

由于 Node 端要到新版对于 ES6 支持较好，所以先用 `babel-register` 在 `src/server/index.js` 去即时转译 `server.js`，但目前不建议在 `production` 环境使用。

```
// use babel-register to precompile ES6 syntax
require('babel-register');
require('./server');
```

现在我们来设计一下我们数据库的 Schema，在这边我们使用 MongoDB 的 ORM Mongoose，可以方便我们使用物件方式进行资料库的操作：

```
// 引入 mongoose 和 Schema
import mongoose, { Schema } from 'mongoose';

// 使用 mongoose.model 建立新的资料表，并将 Schema 传入
// 这边我们设计了食谱分享的一些基本要素，包括名称、描述、照片位置等
export default mongoose.model('Recipe', new Schema({
    id: String,
    name: String,
    description: String,
    imagePath: String,
    steps: Array,
    updatedAt: Date,
}));
```

```
// 引入 mongoose 和 Schema
import mongoose, { Schema } from 'mongoose';

// 使用 mongoose.model 建立新的数据表，并将 Schema 传入
// 这边我们设计了使用者的一些基本要素，包括名称、描述、照片位置等
export default mongoose.model('User', new Schema({
    id: Number,
    username: String,
    email: String,
    password: String,
    admin: Boolean
}));
```

为了方便维护，我们把 API 的部份统一在 `src/server/controllers/api.js` 进行管理，这部份会涉及比较多 Node 和 mongoose 的操作，若读者尚不熟悉可以参考 mongoose 官网

```
import Express from 'express';
// 引入 jsonwebtoken 套件
import jwt from 'jsonwebtoken';
// 引入 User、Recipe Model 方便进行资料库操作
import User from '../models/user';
import Recipe from '../models/recipe';
```

```
import config from '../config';

// API Route
const app = new Express();
const apiRoutes = Express.Router();
// 设定 JSON Web Token 的 secret variable
app.set('superSecret', config.secret); // secret variable
// 使用者登录 API ，依据使用 email 和 密码去验证，若成功则回传一个认证 to
ken（时效24小时）我们把它存在 cookie 中，方便前后端存取。这边我们先不考虑
太多信息安全的问题
apiRoutes.post('/login', function(req, res) {
  // find the user
  User.findOne({
    email: req.body.email
  }, (err, user) => {
    if (err) throw err;
    if (!user) {
      res.json({ success: false, message: 'Authentication failed
. User not found.' });
    } else if (user) {
      // check if password matches
      if (user.password != req.body.password) {
        res.json({ success: false, message: 'Authentication fail
ed. Wrong password.' });
      } else {
        // if user is found and password is right
        // create a token
        const token = jwt.sign({ email: user.email }, app.get('s
uperSecret'), {
          expiresIn: 60 * 60 * 24 // expires in 24 hours
        });
        // return the information including token as JSON
        // 若登录成功回传一个 json 讯息
        res.json({
          success: true,
          message: 'Enjoy your token!',
          token: token,
          userId: user._id
        });
      }
```

```
    }
  });
});
// 初始化 api，一开始数据库尚未建立任何使用者，我们需要在浏览器输入 `http:
//localhost:3000/api/setup`，进行数据库初始化。这个动作将新增一个使用者
、一份食谱，若是成功新增将回传一个 success 讯息
apiRoutes.get('/setup', (req, res) => {
  // create a sample user
  const sampleUser = new User({
    username: 'mark',
    email: 'mark@demo.com',
    password: '123456',
    admin: true
  });
  const sampleRecipe = new Recipe({
    id: '110ec58a-a0f2-4ac4-8393-c866d813b8d1',
    name: '番茄炒蛋',
    description: '番茄炒蛋，一道非常经典的家常菜料理。虽然看似普通，但每
个家庭都有属于自己家里的不同味道',
    imagePath: 'https://c1.staticflickr.com/6/5011/5510599760_66
68df5a8a_z.jpg',
    steps: ['放入番茄', '打个蛋', '放入少许盐巴', '用心快炒'],
    updatedAt: new Date()
  });
  // save the sample user
  sampleUser.save((err) => {
    if (err) throw err;
    sampleRecipe.save((err) => {
      if (err) throw err;
      console.log('User saved successfully');
      res.json({ success: true });
    })
  });
});
// 回传所有 recipes
apiRoutes.get('/recipes', (req, res) => {
  Recipe.find({}, (err, recipes) => {
    res.status(200).json(recipes);
  })
});
```

```
// route middleware to verify a token
// 接下来的 api 将进行控管，也就是说必须在网址请求中夹带认证 token 才能完成请求
apiRoutes.use((req, res, next) => {
  // check header or url parameters or post parameters for token
  // 确认标头、网址或 post 参数是否含有 token，本范例因为简便使用网址 query 参数
  var token = req.body.token || req.query.token || req.headers['x-access-token'];
  // decode token
  if (token) {
    // verifies secret and checks exp
    jwt.verify(token, app.get('superSecret'), (err, decoded) => {
      if (err) {
        return res.json({ success: false, message: 'Failed to authenticate token.' });
      } else {
        // if everything is good, save to request for use in other routes
        req.decoded = decoded;
        next();
      }
    });
  } else {
    // if there is no token
    // return an error
    return res.status(403).send({
        success: false,
        message: 'No token provided.'
    });
  }
});
// 确认认证是否成功
apiRoutes.get('/authenticate', (req, res) => {
  res.json({
    success: true,
    message: 'Enjoy your token!',
  });
```

```
});
// create recipe 新增食谱
apiRoutes.post('/recipes', (req, res) => {
  const newRecipe = new Recipe({
    name: req.body.name,
    description: req.body.description,
    imagePath: req.body.imagePath,
    steps: ['放入番茄', '打个蛋', '放入少许盐巴', '用心快炒'],
    updatedAt: new Date()
  });
  newRecipe.save((err) => {
    if (err) throw err;
    console.log('User saved successfully');
    res.json({ success: true });
  });
});
// update recipe 根据 _id（mongodb 的 id）更新食谱
apiRoutes.put('/recipes/:id', (req, res) => {
  Recipe.update({ _id: req.params.id }, {
    name: req.body.name,
    description: req.body.description,
    imagePath: req.body.imagePath,
    steps: ['放入番茄', '打个蛋', '放入少许盐巴', '用心快炒'],
    updatedAt: new Date()
  } ,(err) => {
    if (err) throw err;
    console.log('User updated successfully');
    res.json({ success: true });
  });
});
// remove recipe 根据 _id 删除食谱，若成功回传成功讯息
apiRoutes.delete('/recipes/:id', (req, res) => {
  Recipe.remove({ _id: req.params.id }, (err, recipe) => {
    if (err) throw err;
    console.log('remove saved successfully');
    res.json({ success: true });
  });
});
export default apiRoutes;
```

设定整个 App 的 routing，我们主要页面有 `HomePageContainer` 、 `LoginPageContainer` 、 `SharePageContainer` ，值得注意的是我们这边使用 Higher Order Components （Higher Order Components 为一个函数， 接收一个 Component 后在 Class Component 的 render 中 return 回传入的 components）方式去确认使用者是否有登录，若有没登录则不能进入分享食谱页面，反之若已登录也不会再进到登录页面：

```
import React from 'react';
import { Route, IndexRoute } from 'react-router';
import Main from '../components/Main';
import CheckAuth from '../components/CheckAuth';
import HomePageContainer from '../containers/HomePageContainer';
import LoginPageContainer from '../containers/LoginPageContainer'
;
import SharePageContainer from '../containers/SharePageContainer'
;

export default (
  <Route path='/' component={Main}>
    <IndexRoute component={HomePageContainer} />
    <Route path="/login" component={CheckAuth(LoginPageContainer
, 'guest')}/>
    <Route path="/share" component={CheckAuth(SharePageContainer
, 'auth')}/>
  </Route>
);
```

设定行为常数（ `src/constants/actionTypes.js` ）：

```
export const AUTH_START    = "AUTH_START";
export const AUTH_COMPLETE = "AUTH_COMPLETE";
export const AUTH_ERROR    = "AUTH_ERROR";
export const START_LOGOUT    = "START_LOGOUT";
export const CHECK_AUTH    = "CHECK_AUTH";
export const SET_USER    = "SET_USER";
export const SHOW_SPINNER    = "SHOW_SPINNER";
export const HIDE_SPINNER    = "HIDE_SPINNER";
export const SET_UI    = "SET_UI";
export const GET_RECIPES = 'GET_RECIPES';
export const SET_RECIPE = 'SET_RECIPE';
export const ADD_RECIPE = 'ADD_RECIPE';
export const UPDATE_RECIPE = 'UPDATE_RECIPE';
export const DELETE_RECIPE = 'DELETE_RECIPE';
```

设定 `src/actions/recipeActions.js` ，我们这边使用 redux-promise，可以很容易使用非同步的行为 WebAPI：

```
import { createAction } from 'redux-actions';
import WebAPI from '../utils/WebAPI';

import {
  GET_RECIPES,
  ADD_RECIPE,
  UPDATE_RECIPE,
  DELETE_RECIPE,
  SET_RECIPE,
} from '../constants/actionTypes';

export const getRecipes = createAction('GET_RECIPES', WebAPI.get
Recipes);
export const addRecipe = createAction('ADD_RECIPE', WebAPI.addRe
cipe);
export const updateRecipe = createAction('UPDATE_RECIPE', WebAPI
.updateRecipe);
export const deleteRecipe = createAction('DELETE_RECIPE', WebAPI
.deleteRecipe);
export const setRecipe = createAction('SET_RECIPE');
```

设定 `src/actions/uiActions.js` ：

```
import { createAction } from 'redux-actions';
import WebAPI from '../utils/WebAPI';

import {
  SHOW_SPINNER,
  HIDE_SPINNER,
  SET_UI,
} from '../constants/actionTypes';

export const showSpinner = createAction('SHOW_SPINNER');
export const hideSpinner = createAction('HIDE_SPINNER');
export const setUi = createAction('SET_UI');
```

设定 `src/actions/userActions.js` ，处理使用者登录登出等行为：

```
import { createAction } from 'redux-actions';
import WebAPI from '../utils/WebAPI';

import {
  AUTH_START,
  AUTH_COMPLETE,
  AUTH_ERROR,
  START_LOGOUT,
  CHECK_AUTH,
  SET_USER
} from '../constants/actionTypes';

export const authStart = createAction('AUTH_START', WebAPI.login
);
export const authComplete = createAction('AUTH_COMPLETE');
export const authError = createAction('AUTH_ERROR');
export const startLogout = createAction('START_LOGOUT', WebAPI.l
ogout);
export const checkAuth = createAction('CHECK_AUTH');
export const setUser = createAction('SET_USER');
```

于 `scr/actions/index.js` 输出 actions：

```
export * from './userActions';
export * from './recipeActions';
export * from './uiActions';
```

于 `scr/common/utils/fetchComponentData.js` 设定 server side 初始 fetchComponentData：

```
// 这边使用 axios 方便进行 promises base request
import axios from 'axios';
// 记得附加上我们存在 cookies 的 token
export default function fetchComponentData(token = 'token') {
  const promises = [axios.get('http://localhost:3000/api/recipes'
), axios.get('http://localhost:3000/api/authenticate?token=' + t
oken)];
  return Promise.all(promises);
}
```

于 `scr/common/utils/WebAPI.js` 所有前端 API 的处理：

```
import axios from 'axios';
import { browserHistory } from 'react-router';
// 引入 uuid 当做食谱 id
import uuid from 'uuid';

import {
  authComplete,
  authError,
  hideSpinner,
  completeLogout,
} from '../actions';

// getCookie 函数传入 key 回传 value
function getCookie(keyName) {
  var name = keyName + '=';
  const cookies = document.cookie.split(';');
```

```
  for(let i = 0; i < cookies.length; i++) {
      let cookie = cookies[i];
      while (cookie.charAt(0)==' ') {
          cookie = cookie.substring(1);
      }
      if (cookie.indexOf(name) == 0) {
        return cookie.substring(name.length, cookie.length);
      }
  }
  return "";
}

export default {
  // 呼叫后端登录 api
  login: (dispatch, email, password) => {
    axios.post('/api/login', {
      email: email,
      password: password
    })
    .then((response) => {
      if(response.data.success === false) {
        dispatch(authError());
        dispatch(hideSpinner());
        alert('发生错误，请再试一次！');
        window.location.reload();
      } else {
        if (!document.cookie.token) {
          let d = new Date();
          d.setTime(d.getTime() + (24 * 60 * 60 * 1000));
          const expires = 'expires=' + d.toUTCString();
          document.cookie = 'token=' + response.data.token + '; '
 + expires;
          dispatch(authComplete());
          dispatch(hideSpinner());
          browserHistory.push('/');
        }
      }
    })
    .catch(function (error) {
      dispatch(authError());
```

```
      });
    },
    // 呼叫后端登出 api
    logout: (dispatch) => {
      document.cookie = 'token=; ' + 'expires=Thu, 01 Jan 1970 00:
00:01 GMT;';
      dispatch(hideSpinner());
      browserHistory.push('/');
    },
    // 确认使用者是否登录
    checkAuth: (dispatch, token) => {
      axios.post('/api/authenticate', {
        token: token,
      })
      .then((response) => {
        if(response.data.success === false) {
          dispatch(authError());
        } else {
          dispatch(authComplete());
        }
      })
      .catch(function (error) {
        dispatch(authError());
      });
    },
    // 取得目前所有食谱
    getRecipes: () => {
      axios.get('/api/recipes')
      .then((response) => {
      })
      .catch((error) => {
      });
    },
    // 呼叫新增食谱 api，记得附加上我们存在 cookies 的 token
    addRecipe: (dispatch, name, description, imagePath) => {
      const id = uuid.v4();
      axios.post('/api/recipes?token=' + getCookie('token'), {
        id: id,
        name: name,
        description: description,
```

```
        imagePath: imagePath,
      })
      .then((response) => {
        if(response.data.success === false) {
          dispatch(hideSpinner());
          alert('发生错误，请再试一次！');
          browserHistory.push('/share');
        } else {
          dispatch(hideSpinner());
          window.location.reload();
          browserHistory.push('/');
        }
      })
      .catch(function (error) {
      });
    },
    // 呼叫更新食谱 api，记得附加上我们存在 cookies 的 token
    updateRecipe: (dispatch, recipeId, name, description, imagePat
  h) => {
      axios.put('/api/recipes/' + recipeId + '?token=' + getCookie(
  'token'), {
        id: recipeId,
        name: name,
        description: description,
        imagePath: imagePath,
      })
      .then((response) => {
        if(response.data.success === false) {
          dispatch(hideSpinner());
          dispatch(setRecipe({ key: 'recipeId', value: '' }));
          dispatch(setUi({ key: 'isEdit', value: false }));
          alert('发生错误，请再试一次！');
          browserHistory.push('/share');
        } else {
          dispatch(hideSpinner());
          window.location.reload();
          browserHistory.push('/');
        }
      })
      .catch(function (error) {
```

```
      });
    },
    // 呼叫删除食谱 api，记得附加上我们存在 cookies 的 token
    deleteRecipe: (dispatch, recipeId) => {
      axios.delete('/api/recipes/' + recipeId + '?token=' + getCoo
  kie('token'))
      .then((response) => {
        if(response.data.success === false) {
          dispatch(hideSpinner());
          alert('发生错误，请再试一次！');
          browserHistory.push('/');
        } else {
          dispatch(hideSpinner());
          window.location.reload();
          browserHistory.push('/');
        }
      })
      .catch(function (error) {
      });
    }
  };
```

接下来设定我们的 `reducers`，以下是 `src/common/reducers/data/recipeReducers.js`，`GET_RECIPES` 负责将后端 API 取得的所有食谱存放在 recipes 中：

```
import { handleActions } from 'redux-actions';
import { RecipeState } from '../../constants/models';

import {
  GET_RECIPES,
  SET_RECIPE,
} from '../../constants/actionTypes';

const recipeReducers = handleActions({
  GET_RECIPES: (state, { payload }) => (
    state.set(
      'recipes',
      payload.recipes
    )
  ),
  SET_RECIPE: (state, { payload }) => (
    state.setIn(payload.keyPath, payload.value)
  ),
}, RecipeState);

export default recipeReducers;
```

以下是 `src/common/reducers/data/userReducers.js` ，负责确认登录相关处理事项。注意的是由于登录是非同步执行，所以会有几个阶段的行为要做处理：

```
import { handleActions } from 'redux-actions';
import { UserState } from '../../constants/models';

import {
  AUTH_START,
  AUTH_COMPLETE,
  AUTH_ERROR,
  LOGOUT_START,
  SET_USER,
} from '../../constants/actionTypes';

const userReducers = handleActions({
  AUTH_START: (state) => (
    state.merge({
```

```
      isAuthorized: false,
    })
  ),
  AUTH_COMPLETE: (state) => (
    state.merge({
      email: '',
      password: '',
      isAuthorized: true,
    })
  ),
  AUTH_ERROR: (state) => (
    state.merge({
      username: '',
      email: '',
      password: '',
      isAuthorized: false,
    })
  ),
  START_LOGOUT: (state) => (
    state.merge({
      isAuthorized: false,
    })
  ),
  CHECK_AUTH: (state) => (
    state.set('isAuthorized', true)
  ),
  SET_USER: (state, { payload }) => (
    state.set(payload.key, payload.value)
  ),
}, UserState);

export default userReducers;
```

以下是 `src/common/reducers/ui/uiReducers.js`，负责确认 UI State 相关处理：

```
import { handleActions } from 'redux-actions';
import { UiState } from '../../constants/models';

import {
  SHOW_SPINNER,
  HIDE_SPINNER,
  SET_UI,
} from '../../constants/actionTypes';

const uiReducers = handleActions({
  SHOW_SPINNER: (state) => (
    state.set(
      'spinnerVisible',
      true
    )
  ),
  HIDE_SPINNER: (state) => (
    state.set(
      'spinnerVisible',
      false
    )
  ),
  SET_UI: (state, { payload }) => (
    state.set(payload.key, payload.value)
  ),
}, UiState);

export default uiReducers;
```

最后把所有 recipes 在 `src/common/reducers/index.js` 使用 `combineReducers` 整合在一起，注意的是我们整个 App 的资料流要维持 immutable：

```
import { combineReducers } from 'redux-immutable';
import ui from './ui/uiReducers';
import recipe from './data/recipeReducers';
import user from './data/userReducers';
// import routes from './routes';

const rootReducer = combineReducers({
  ui,
  recipe,
  user,
});

export default rootReducer;
```

以下是 `src/common/store/configureStore.js` 处理 store 的建立，这次我们使用了 `promiseMiddleware` 的 middleware：

```
import { createStore, applyMiddleware } from 'redux';
import promiseMiddleware from 'redux-promise';
import createLogger from 'redux-logger';
import Immutable from 'immutable';
import rootReducer from '../reducers';

const initialState = Immutable.Map();

export default function configureStore(preloadedState = initialS
tate) {
  const store = createStore(
    rootReducer,
    preloadedState,
    applyMiddleware(createLogger({ stateTransformer: state => st
ate.toJS() }, promiseMiddleware))
  );

  return store;
}
```

经过一连串努力，我们来到了 View 的布建。在这个 App 中我们主要会由一个 AppBar 负责所有页面的导览，也就是每个页面都会有 AppBar 常驻在上面，然而上面的内容则会依 UI State 中的 isAuthorized 而有所不同。最后要留意的是我们使用了 React Bootstrapt 来建立 React Component。

```
import React from 'react';
import { LinkContainer } from 'react-router-bootstrap';
import { Link } from 'react-router';
import { Navbar, Nav, NavItem, NavDropdown, MenuItem } from 'rea
ct-bootstrap';

const AppBar = ({
  isAuthorized,
  onToShare,
  onLogout,
}) => (
  <Navbar>
    <Navbar.Header>
      <Navbar.Brand>
        <Link to="/">OpenCook</Link>
      </Navbar.Brand>
      <Navbar.Toggle />
    </Navbar.Header>
    <Navbar.Collapse>
      {
        isAuthorized === false ?
        (
          <Nav pullRight>
            <LinkContainer to={{ pathname: '/login' }}><NavItem
eventKey={2} href="#">登录</NavItem></LinkContainer>
          </Nav>
        ) :
        (
          <Nav pullRight>
            <NavItem eventKey={1} onClick={onToShare}>分享食谱</N
avItem>
            <NavItem eventKey={2} onClick={onLogout} href="#">登
出</NavItem>
          </Nav>
```

```
        )
      }
    </Navbar.Collapse>
  </Navbar>
);

export default AppBar;
```

以下是 `src/common/containers/AppBarContainer/AppBarContainer.js` ：

```
import React from 'react';
import { connect } from 'react-redux';
import AppBar from '../../components/AppBar';
import { browserHistory } from 'react-router';

import {
  startLogout,
  setRecipe,
  setUi,
} from '../../actions';

export default connect(
  (state) => ({
    isAuthorized: state.getIn(['user', 'isAuthorized']),
  }),
  (dispatch) => ({
    onToShare: () => {
      dispatch(setRecipe({ key: 'recipeId', value: '' }));
      dispatch(setUi({ key: 'isEdit', value: false }));
      window.location.reload();
      browserHistory.push('/share');
    },
    onLogout: () => (
      dispatch(startLogout(dispatch))
    ),
  })
)(AppBar);
```

以下是 `src/components/Main/Main.js` ，透过 route 机制让 AppBarContainer 可以成为整个 App 母模版：

```
import React from 'react';
import AppBarContainer from '../../containers/AppBarContainer';

const Main = (props) => (
  <div>
    <AppBarContainer />
    <div>
      {props.children}
    </div>
  </div>
);

export default Main;
```

在 `checkAuth` 这个 Component 中，我们使用到了 Higher Order Components 的观念。Higher Order Components 为一个函数， 接收一个 Component 后在 Class Component 的 render 中 return 回传入的 components 方式去确认使用者是否有登录，若有没登录则不能进入分享食谱页面，反之若已登录也不会再进到登录页面：

```
import React from 'react';
import { connect } from 'react-redux';
import { withRouter } from 'react-router';

// High Order Component
export default function requireAuthentication(Component, type) {
  class AuthenticatedComponent extends React.Component {
    componentWillMount() {
      this.checkAuth();
    }
    componentWillReceiveProps(nextProps) {
      this.checkAuth();
    }
    checkAuth() {
      if(type === 'auth') {
```

```
    if (!this.props.isAuthorized) {
     this.props.router.push('/');
    }
   } else {
    if (this.props.isAuthorized) {
     this.props.router.push('/');
    }
   }
  }
  render() {
   return (
    <div>
    {
     (type === 'auth') ?
     this.props.isAuthorized === true ? <Component {...this
.props } /> : null
     : this.props.isAuthorized === false ? <Component {...t
his.props } /> : null
    }
    </div>
   )
  }
 };
 const mapStateToProps = (state) => ({
  isAuthorized: state.getIn(['user', 'isAuthorized']),
 });
 return connect(mapStateToProps)(withRouter(AuthenticatedCompon
ent));
}
```

我们将每个食谱呈现设计成 RecipeBox，以下是在 `src/common/components/HomePage/HomePage.js` 使用 map 方法去迭代我们的食谱：

```
import React from 'react';
import RecipeBoxContainer from '../../containers/RecipeBoxContai
ner';

const HomePage = ({
  recipes
}) => (
  <div>
  {
    recipes.map((recipe, index) => (
      <RecipeBoxContainer recipe={recipe} key={index}  />
    )).toJS()
  }
  </div>
);

export default HomePage;
```

以下是 `src/common/containers/HomePageContainer/HomePageContainer.js`：

```
import React from 'react';
import { connect } from 'react-redux';
import HomePage from '../../components/HomePage';

export default connect(
  (state) => ({
    recipes: state.getIn(['recipe', 'recipes']),
  }),
  (dispatch) => ({
  })
)(HomePage);
```

在 `src/common/components/LoginBox/LoginBox.js` 设计我们 LoginBox：

```
import React from 'react';
import { Form, FormGroup, Button, FormControl, ControlLabel } fr
om 'react-bootstrap';
```

```
const LoginBox = ({
  email,
  password,
  onChangeEmailInput,
  onChangePasswordInput,
  onLoginSubmit
}) => (
  <div>
    <Form horizontal>
      <FormGroup
        controlId="formBasicText"
      >
        <ControlLabel>请输入您的 Email</ControlLabel>
        <FormControl
          type="text"
          onChange={onChangeEmailInput}
          placeholder="Enter Email"
        />
        <FormControl.Feedback />
      </FormGroup>
      <FormGroup
        controlId="formBasicText"
      >
        <ControlLabel>请输入您的密码</ControlLabel>
        <FormControl
          type="password"
          onChange={onChangePasswordInput}
          placeholder="Enter Password"
        />
        <FormControl.Feedback />
      </FormGroup>
      <Button
        onClick={onLoginSubmit}
        bsStyle="success"
        bsSize="large"
        block
      >
        提交送出
      </Button>
```

```
    </Form>
  </div>
);

export default LoginBox;
```

以下是 `src/common/containers/LoginBoxContainer/LoginBoxContainer.js`：

```
import React from 'react';
import { connect } from 'react-redux';
import LoginBox from '../../components/LoginBox';

import {
  authStart,
  showSpinner,
  setUser,
} from '../../actions';

export default connect(
  (state) => ({
    email: state.getIn(['user', 'email']),
    password: state.getIn(['user', 'password']),
  }),
  (dispatch) => ({
    onChangeEmailInput: (event) => (
      dispatch(setUser({ key: 'email', value: event.target.value
 }))
    ),
    onChangePasswordInput: (event) => (
      dispatch(setUser({ key: 'password', value: event.target.va
lue }))
    ),
    onLoginSubmit: (email, password) => () => {
      dispatch(authStart(dispatch, email, password));
      dispatch(showSpinner());
    },
  }),
  (stateProps, dispatchProps, ownProps) => {
    const { email, password } = stateProps;
    const { onLoginSubmit } = dispatchProps;
    return Object.assign({}, stateProps, dispatchProps, ownProps
, {
      onLoginSubmit: onLoginSubmit(email, password),
    });
  }
)(LoginBox);
```

在 `src/common/components/LoginPage/LoginPage.js` ，当 spinnerVisible 为 true 会显示 spinner：

```
import React from 'react';
import { Grid, Row, Col, Image } from 'react-bootstrap';
import LoginBoxContainer from '../../containers/LoginBoxContaine
r';

const LoginPage = ({
  spinnerVisible,
}) => (
  <div>
    <Row className="show-grid">
      <Col xs={6} xsOffset={3}>
        <LoginBoxContainer />
        { spinnerVisible === true ?
          <Image src="/static/images/loading.gif" /> :
          null
        }
      </Col>
    </Row>
  </div>
);

export default LoginPage;
```

以下是 `src/common/containers/LoginPageContainer/LoginPageContainer.js` ：

```
import React from 'react';
import { connect } from 'react-redux';
import LoginPage from '../../components/LoginPage';

export default connect(
  (state) => ({
    spinnerVisible: state.getIn(['ui', 'spinnerVisible']),
  }),
  (dispatch) => ({
  })
)(LoginPage);
```

真正设计我们内部的食谱， `src/common/components/RecipeBox` ，使用者登录的话可以修改和删除食谱：

```
import React from 'react';
import { Grid, Row, Col, Image, Thumbnail, Button } from 'react-
bootstrap';

const RecipeBox = (props) => {
  return(
      <Col xs={6} md={4}>
        <Thumbnail src={props.recipe.get('imagePath')} alt="242x
200">
          <h3>{props.recipe.get('name')}</h3>
          <p>{props.recipe.get('description')}</p>
          {
            props.isAuthorized === true ? (
            <p>
              <Button bsStyle="primary" onClick={props.onDeleteR
ecipe(props.recipe.get('_id'))}>删除</Button> 
              <Button bsStyle="default" onClick={props.onUpadate
Recipe(props.recipe.get('_id'))}>修改</Button>
            </p>)
            : null
          }
        </Thumbnail>
      </Col>
    );
}

export default RecipeBox;
```

以下是 `src/common/containers/RecipeBoxContainer/RecipeBoxContainer.js`：

```
import React from 'react';
import { connect } from 'react-redux';
import RecipeBox from '../../components/RecipeBox';
import { browserHistory } from 'react-router';

import {
  deleteRecipe,
  setRecipe,
```

```
  setUi
} from '../../actions';

export default connect(
  (state) => ({
    isAuthorized: state.getIn(['user', 'isAuthorized']),
    recipes: state.getIn(['recipe', 'recipes']),
  }),
  (dispatch) => ({
    onDeleteRecipe: (recipeId) => () => (
      dispatch(deleteRecipe(dispatch, recipeId))
    ),
    onUpadateRecipe: (recipes) => (recipeId) => () => {
      const recipeIndex = recipes.findIndex((_recipe) => (_recip
e.get('_id') === recipeId));
      const recipe = recipeIndex !== -1 ? recipes.get(recipeInde
x) : undefined;
      dispatch(setRecipe({ keyPath: ['recipe'], value: recipe })
);
      dispatch(setRecipe({ keyPath: ['recipe', 'id'], value: rec
ipeId }));
      dispatch(setUi({ key: 'isEdit', value: true }));
      browserHistory.push('/share?recipeId=' + recipeId);
    },
  }),
  (stateProps, dispatchProps, ownProps) => {
    const { recipes } = stateProps;
    const { onUpadateRecipe } = dispatchProps;
    return Object.assign({}, stateProps, dispatchProps, ownProps
, {
      onUpadateRecipe: onUpadateRecipe(recipes),
    });
  }
)(RecipeBox);
```

设计我们分享食谱页面，这边我们把编辑食谱和新增分享一起共用了同一个 components，差别在于我们会判断 UI State 中的 `isEdit` ，决定相应处理方式。在中 `src/common/components/ShareBox/ShareBox.js` ，可以让使用者登录的后修改和删除食谱：

```
import React from 'react';
import { Form, FormGroup, Button, FormControl, ControlLabel } fr
om 'react-bootstrap';

const ShareBox = (props) => {
  return (<div>
    <Form horizontal>
      <FormGroup
        controlId="formBasicText"
      >
        <ControlLabel>请输入食谱名称</ControlLabel>
        <FormControl
          type="text"
          placeholder="Enter text"
          defaultValue={props.name}
          onChange={props.onChangeNameInput}
        />
        <FormControl.Feedback />
      </FormGroup>
      <FormGroup
        controlId="formBasicText"
      >
        <ControlLabel>请输入食谱说明</ControlLabel>
        <FormControl
          componentClass="textarea"
          placeholder="textarea"
          defaultValue={props.description}
          onChange={props.onChangeDescriptionInput}
        />
        <FormControl.Feedback />
      </FormGroup>
      <FormGroup
        controlId="formBasicText"
      >
        <ControlLabel>请输入食谱图片网址</ControlLabel>
        <FormControl
          type="text"
          placeholder="Enter text"
          defaultValue={props.imagePath}
          onChange={props.onChangeImageUrl}
```

```
          />
          <FormControl.Feedback />
        </FormGroup>
        <Button
          onClick={props.onRecipeSubmit}
          bsStyle="success"
          bsSize="large"
          block
        >
          提交送出
        </Button>
      </Form>
    </div>);
};

export default ShareBox;
```

以下是 `src/common/containers/ShareBoxContainer/ShareBoxContainer.js`：

```
import React from 'react';
import { connect } from 'react-redux';
import ShareBox from '../../components/ShareBox';

import {
  addRecipe,
  updateRecipe,
  showSpinner,
  setRecipe,
} from '../../actions';

export default connect(
  (state) => ({
    recipes: state.getIn(['recipe', 'recipes']),
    recipeId: state.getIn(['recipe', 'recipe', 'id']),
    name: state.getIn(['recipe', 'recipe', 'name']),
    description: state.getIn(['recipe', 'recipe', 'description']
),
    imagePath: state.getIn(['recipe', 'recipe', 'imagePath']),
```

```
    isEdit: state.getIn(['ui', 'isEdit']),
  }),
  (dispatch) => ({
    onChangeNameInput: (event) => (
      dispatch(setRecipe({ keyPath: ['recipe', 'name'], value: e
vent.target.value }))
    ),
    onChangeDescriptionInput: (event) => (
      dispatch(setRecipe({ keyPath: ['recipe', 'description'], v
alue: event.target.value }))
    ),
    onChangeImageUrl: (event) => (
      dispatch(setRecipe({ keyPath: ['recipe', 'imagePath'], val
ue: event.target.value }))
    ),
    onRecipeSubmit: (recipes, recipeId, name, description, image
Path, isEdit) => () => {
      if (isEdit === true) {
        dispatch(updateRecipe(dispatch, recipeId, name, descript
ion, imagePath));
        dispatch(showSpinner());
      } else {
        dispatch(addRecipe(dispatch, name, description, imagePat
h));
        dispatch(showSpinner());
      }
    },
  }),
  (stateProps, dispatchProps, ownProps) => {
    const { recipes, recipeId, name, description, imagePath, isE
dit } = stateProps;
    const { onRecipeSubmit } = dispatchProps;
    return Object.assign({}, stateProps, dispatchProps, ownProps
, {
      onRecipeSubmit: onRecipeSubmit(recipes, recipeId, name, de
scription, imagePath, isEdit),
    });
  }
)(ShareBox);
```

单纯的 SharePage（ `src/common/components/SharePage/SharePage.js` ）页面：

```
import React from 'react';
import { Grid, Row, Col } from 'react-bootstrap';
import ShareBoxContainer from '../../containers/ShareBoxContaine
r';

const SharePage = () => (
  <div>
    <Row className="show-grid">
      <Col xs={6} xsOffset={3}>
        <ShareBoxContainer />
      </Col>
    </Row>
  </div>
);

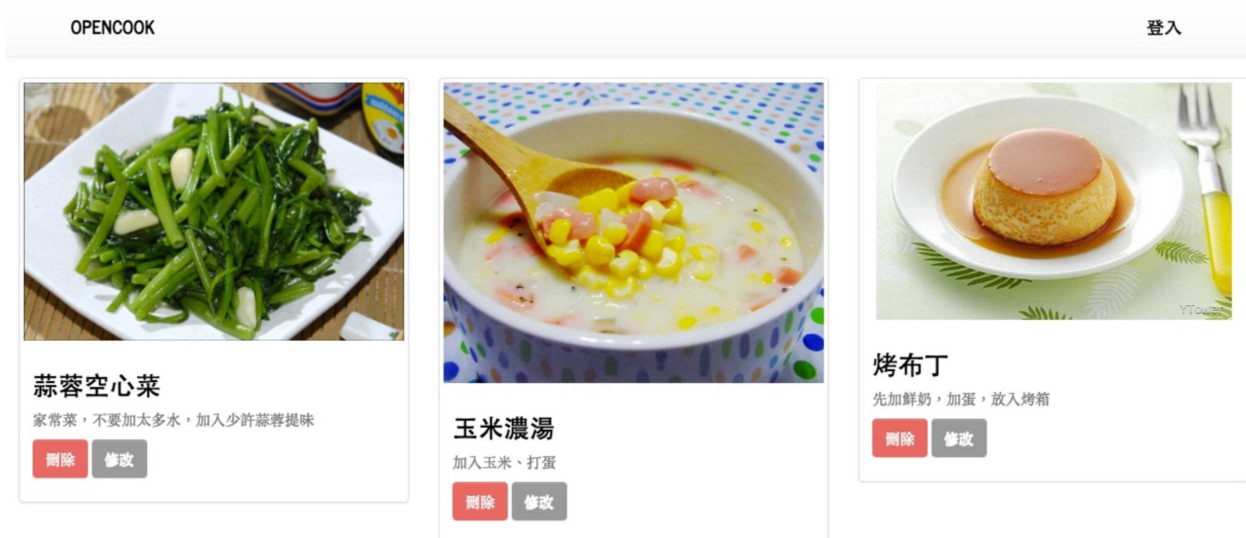
export default SharePage;
```

以下是 `src/common/containers/SharePageContainer/SharePageContainer.js` ：

```
import React from 'react';
import { connect } from 'react-redux';
import SharePage from '../../components/SharePage';

export default connect(
  (state) => ({
  }),
  (dispatch) => ({
  })
)(SharePage);
```

恭喜你成功抵达终点！若一切顺利，在终端机打上 `$ npm start` ，你将可以在浏览器的 `http://localhost:3000` 看到自己的成果！



总结

本章整合过去所学和添加一些后端资料库知识开发了一个可以登录会员并分享食谱的社群网站！快把你的成果和你的朋友分享吧！觉得意犹未尽？别忘了附录也很精采！最后，再次谢谢读者们支持我们一路走完了 React 开发学习之旅！然而前端技术变化很快，唯有不断自我学习才能持续成长。笔者才疏学浅，撰写学习心得或有疏漏，若有任何建议或提醒都欢迎和我说，大家一起加油：）

延伸阅读

1. joshgeller/react-redux-jwt-auth-example
2. Securing React Redux Apps With JWT Tokens
3. Adding Authentication to Your React Native App Using JSON Web Tokens
4. Authentication in React Applications, Part 2: JSON Web Token (JWT)
5. Node.js 身份认证：Passport 入门
6. react-bootstrap compatibility #83
7. How to authenticate routes using Passport? #725
8. Isomorphic React Web App Demo with Material UI
9. react-router/examples/auth-flow/
10. redux-promise
11. How to use redux-promise
12. Authenticate a Node.js API with JSON Web Tokens

13. 3 JavaScript ORMs You Might Not Know
14. lynndylanhurley/redux-auth
15. How to avoid getting error 'localStorage is not defined' on server in ReactJS isomorphic app?
16. Where to Store your JWTs – Cookies vs HTML5 Web Storage
17. What is the difference between server side cookie and client side cookie? [closed]
18. Cookies vs Tokens. Getting auth right with Angular.JS
19. Cookies vs Tokens: The Definitive Guide
20. joshgeller/react-redux-jwt-auth-example
21. Programmatically navigate using react router
22. withRouter HoC (higher-order component) v2.4.0 Upgrade Guide

License

:door: 任意门

| 回首页 | 上一章：React Redux Sever Rendering（Isomorphic JavaScript）入门 | 下一章：附录一、React ES5、ES6+ 常见用法对照表 |

| 纠错、提问或许愿 |

附录一、React ES5、ES6+ 常见用法对照表



前言

React 是 Facebook 推出的开源 JavaScript Library。自从 React 正式开源后，React 生态系开始蓬勃发展。事实上，透过学习 React 生态系（ecosystem）的过程中，可以上我们顺便学习现代化 Web 开发的重要观念（例如：ES6、Webpack、Babel、组件化等），成为更好的开发者。虽然 ES6（ECMAScript2015）、ES7 是未来趋势（本文將 ES6、ES7 称为 ES6+），然而目前在网络上有许多的学习资源仍是以 ES5 为主，导致读者在学习上遇到一些坑和迷惑（本文假设读者对于 React 已经有些基本认识，若你对于 React 尚不熟悉，建议先行阅读官方文件和本篇入门教学）。因此本文希望透過整理在 React 中 ES5、ES6+ 常见用法对照表，让读者们可以在实现功能时（尤其在 React Native）可以更清楚两者的差异，无痛转移到 ES6+。

大纲

1. Modules
2. Classes
3. Method definition
4. Property initializers
5. State
6. Arrow functions
7. Dynamic property names & template strings

8. Destructuring & spread attributes
9. Mixins
10. Default Parameters

1. Modules

随着 Web 技术的进展，组件化开发已经成为一个重要课题。关于 JavaScript 组件化我们这边不详述，建议读者参考 PPT 和 这篇文章。

ES5 若使用 CommonJS 标准，一般使用 `require()` 用法引入模块：

```
var React = require('react');
var MyComponent = require('./MyComponent');
```

输出则是使用 `module.exports` ：

```
module.exports = MyComponent;
```

ES6+ `import` 用法：

```
import React from 'react';
import MyComponent from './MyComponent';
```

输出则是使用 `export default` ：

```
export default class MyComponent extends React.Component {

}
```

2. Classes

在 React 中组件（Component）是组成视觉页面的基础。在 ES5 中我们使用 `React.createClass()` 来建立 Component，而在 ES6+ 则是用 Classes 继承 `React.Component` 来建立 Component。若是有写过 Java 等面向对象语言（OOP）的读者应该对于这种写法比较不陌生，不过要注意的是 JavaScript 仍是

原型继承类型的面向对象程序语言，只是使用 `Classes` 让面向对象使用上更加直观。对于选择 `class` 使用上还有疑惑的读者建议可以阅读 React.createClass versus extends React.Component 这篇文章。

ES5 `React.createClass()` 用法：

```
var Photo = React.createClass({
  render: function() {
    return (
      <div>
        <images alt={this.props.description} src={this.props.src
} />
      </div>
      );
  }
});
ReactDOM.render(<Photo />, document.getElementById('main'));
```

ES6+ `class` 用法：

```
class Photo extends React.Component {
  render() {
    return <images alt={this.props.description} src={this.props.
src} />;
  }
}
ReactDOM.render(<Photo />, document.getElementById('main'));
```

在 ES5 我們会在 `componentWillMount` 生命周期定义希望在 `render` 前执行，且只会执行一次的任务：

```
var Photo = React.createClass({
  componentWillMount: function() {}
});
```

在 ES6+ 则是定义在 `constructor` 建构子中：

```
class Photo extends React.Component {
  constructor(props) {
    super(props);
    // 原本在 componentWillMount 操作的动作可以放在这
  }
}
```

3. Method definition

在 ES6 中我们使用 `Method` 可以忽略 `function` 和 `,`，使用上更为简洁！ES5 `React.createClass()` 用法：

```
var Photo = React.createClass({
  handleClick: function(e) {},
  render: function() {}
});
```

ES6+ class 用法：

```
class Photo extends React.Component {
  handleClick(e) {}
  render() {}
}
```

4. Property initializers

Component 属性值是数据传递重要的元素，在 ES5 中我们使用 `propTypes` 和 `getDefaultProps` 来定义属性（props）的预设值和型别：

```
var Todo = React.createClass({
  getDefaultProps: function() {
    return {
      checked: false,
      maxLength: 10,
    };
  },
  propTypes: {
    checked: React.PropTypes.bool.isRequired,
    maxLength: React.PropTypes.number.isRequired
  },
  render: function() {
    return();
  }
});
```

在 ES6+ 中我们则是参考 ES7 property initializers 使用 `class` 中的静态属性（static properties）来定义：

```
class Todo extends React.Component {
  static defaultProps = {
    checked: false,
    maxLength: 10,
  }; // 注意有分号
  static propTypes = {
    checked: React.PropTypes.bool.isRequired,
    maxLength: React.PropTypes.number.isRequired
  };
  render() {
    return();
  }
}
```

ES6+ 另外一种写法，可以留意一下，主要是看各团队喜好和规范，选择合适的方式：

```
class Todo extends React.Component {
    render() {
        return (
            <View />
        );
    }
}
Todo.defaultProps = {
    checked: false,
    maxLength: 10,
};
Todo.propTypes = {
    checked: React.PropTypes.bool.isRequired,
    maxLength: React.PropTypes.number.isRequired,
};
```

5. State

在 React 中 `Props` 和 `State` 是数据流传递的重要元素，不同的是 `state` 可更改，可以去执行一些运算。在 ES5 中我们使用 `getInitialState` 去初始化 `state`：

```
var Todo = React.createClass({
    getInitialState: function() {
        return {
            maxLength: this.props.maxLength,
        };
    },
});
```

在 ES6+ 中我们初始化 `state` 有两种写法：

```
class Todo extends React.Component {
    state = {
        maxLength: this.props.maxLength,
    }
}
```

另外一种写法，使用在构造函数初始化。比较推荐使用这种方式，方便做一些运算：

```
class Todo extends React.Component {
    constructor(props){
        super(props);
        this.state = {
            maxLength: this.props.maxLength,
        };
    }
}
```

6. Arrow functions

在讲 `Arrow functions` 之前，我们先聊聊在 React 中 `this` 和它所代表的 `context`。在 ES5 中，我们使用 `React.createClass()` 来建立 Component，而在 `React.createClass()` 下，预设帮你绑定好 `method` 的 `this`，你毋须自行绑定。所以你可以看到像是下面的例子，`callback function` handleButtonClick 中的 `this` 是指到 component 的实例（instance），而非触发事件的对象：

```
var TodoBtn = React.createClass({
    handleButtonClick: function(e) {
        // 此 this 指到 component 的实例（instance），而非 button
        this.setState({showOptionsModal: true});
    },
    render: function(){
        return (
            <div>
                <Button onClick={this.handleButtonClick}>{this.p
rops.label}</Button>
            </div>
        )
    },
});
```

然而自动绑定这种方式反而会让人容易误解，所以在 ES6+ 推荐使用 `bind` 绑定 `this` 或使用 `Arrow functions` （它会绑定当前 `scope` 的 `this` `context` ）两种方式，你可以参考下面例子：

```
class TodoBtn extends React.Component
{
    handleButtonClick(e){
        // 确认绑定 this 指到 component instance
        this.setState({toggle: true});
    }
    render(){
        // 这边可以用 this.handleButtonClick.bind(this) 手动绑定或是
 Arrow functions () => {} 用法
        return (
            <div>
                <Button onClick={this.handleButtonClick.bind(thi
s)} onClick={(e)=> {this.handleButtonClick(e)} }>{this.props.lab
el}</Button>
            </div>
        )
    },
}
```

`Arrow functions` 虽然一开始看起来有点怪异，但其实观念很简单：一个简化的函数。函数基本上就是参数（不一定要有参数）、表达式、回传值（也可能是回传 undefined）：

```
// Arrow functions 的一些例子
()=>7
e=>e+2
()=>{
    alert('XD');
}
(a,b)=>a+b
e=>{
    if (e == 2){
        return 2;
    }
    return 100/e;
}
```

不过要注意的是无论是 `bind` 或是 `Arrow functions`，每次执行回传都是指到一个新的函数，若需要再调用到这个函数，请记得先把它存起来：

错误用法：

```
class TodoBtn extends React.Component{
    componentWillMount(){
        Btn.addEventListener('click', this.handleButtonClick.bin
d(this));
    }
    componentDidmount(){
        Btn.removeEventListener('click', this.handleButtonClick.
bind(this));
    }
    onAppPaused(event){
    }
}
```

正确用法：

```
class TodoBtn extends React.Component{
    constructor(props){
        super(props);
        this.handleButtonClick = this.handleButtonClick.bind(this
);
    }
    componentWillMount(){
        Btn.addEventListener('click', this.handleButtonClick);
    }
    componentDidMount(){
        Btn.removeEventListener('click', this.handleButtonClick)
;
    }
}
```

更多 Arrows and Lexical This 特性可以参考这个文件。

7. Dynamic property names & template strings

以前在 ES5 我们要动态设定属性名称时，往往需要多写几行代码才能达到目标：

```
var Todo = React.createClass({
  onChange: function(inputName, e) {
    var stateToSet = {};
    stateToSet[inputName + 'Value'] = e.target.value;
    this.setState(stateToSet);
  },
});
```

但在 ES6+中，透过 enhancements to object literals 和 template strings 可以轻松完成动态设定属性名称的任务：

```
class Todo extends React.Component {
  onChange(inputName, e) {
    this.setState({
      [`${inputName}Value`]: e.target.value,
    });
  }
}
```

Template Strings 是一种语法糖（syntactic sugar），方便我们组织字串（这边也用上 `let`、`const` 变数和常数定义的方式，和 `var` 的 `function scope` 不同的是它们是属于 `block scope`，亦即作用域存在于 `{}` 间）：

```
// Interpolate variable bindings
const name = "Bob", let = "today";
`Hello ${name}, how are you ${time}?` \\ Hello Bob, how are you today?
```

8. Destructuring & spread attributes

在 React 的 Component 中，父组件利用 `props` 来传递数据到子组件是常见作法，然而我们有时会希望只传递部分数据，此时 ES6+ 中的 Destructuring 和 JSX 的 Spread Attributes，`...` Spread Attributes 主要是用来迭代对象：

```
class Todo extends React.Component {
  render() {
    var {
      className,
      ...others,  // ...others 包含 this.props 除了 className 外所有值。this.props = {value: 'true', title: 'header', className: 'content'}
    } = this.props;
    return (
      <div className={className}>
        <TodoList {...others} />
        <button onClick={this.handleLoadMoreClick}>Load more</button>
      </div>
    );
  }
}
```

但使用上要注意的是若是有重复的属性值则以后来覆盖，下面的例子中若 `...this.props` ，有 `className` ，则被后来的 `main` 所覆盖：

```
<div {...this.props} className="main">
  …
</div>
```

而 `Destructuring` 也可以用在简化 `Module` 的引入上，这边我们先用 ES5 中引入方式来看：

```
var React = require('react-native');
var Component = React.component;

class HelloWorld extends Component {
  render() {
    return (
      <View>
        <Text>Hello, world!</Text>
      </View>
    );
  }
}

export default HelloWorld;
```

以下 ES5 写法：

```
var React = require('react-native');
var View = React.View;
```

在 ES6+ 则可以直接使用 `Destructuring` 这种简化方式来引入模块中的组件：

```
// 这边等于上面的写法
var { View } = require('react-native');
```

更进一步可以使用 `import` 语法：

```
import React, {
  View,
  Component,
  Text,
} from 'react-native';

class HelloWorld extends Component {
  render() {
    return (
      <View>
        <Text>Hello, world!</Text>
      </View>
    );
  }
}

export default HelloWorld;
```

9. Mixins

在 ES5 中，我们可以使用 `Mixins` 的方式去让不同的 Component 共用相似的功能，重用我们的代码：

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

但由于官方不打算在 ES6+ 中继续推行 `Mixins`，若还是希望使用，可以参考看看第三方套件或是这个文件的用法。

10. Default Parameters

以前 ES5 我们函数要使用预设值需要这样使用：

```
var link = function (height, color) {
    var height = height || 50;
    var color = color || 'red';
}
```

现在 ES6+ 的函数可以支援预设值，让代码更为简洁：

```
var link = function(height = 50, color = 'red') {
  ...
}
```

总结

以上就是 React ES5、ES6+常见用法对照表，能看到这边的你应该已经对于 React ES5、ES6 使用上有些认识，先给自己一些掌声吧！确实从 ES6 开始，JavaScript 和以前我们看到的 JavaScript 有些不同，增加了许多新的特性，有些读者甚至会很怀疑说这真的是 JavaScript 吗？ES6 的用法对于初学者来说可能会需要写一点时间吸收，下一章我们将进到同样也是有革新性设计和有趣的 React Native，用 JavaScript 和 React 写 Native App！

延伸阅读

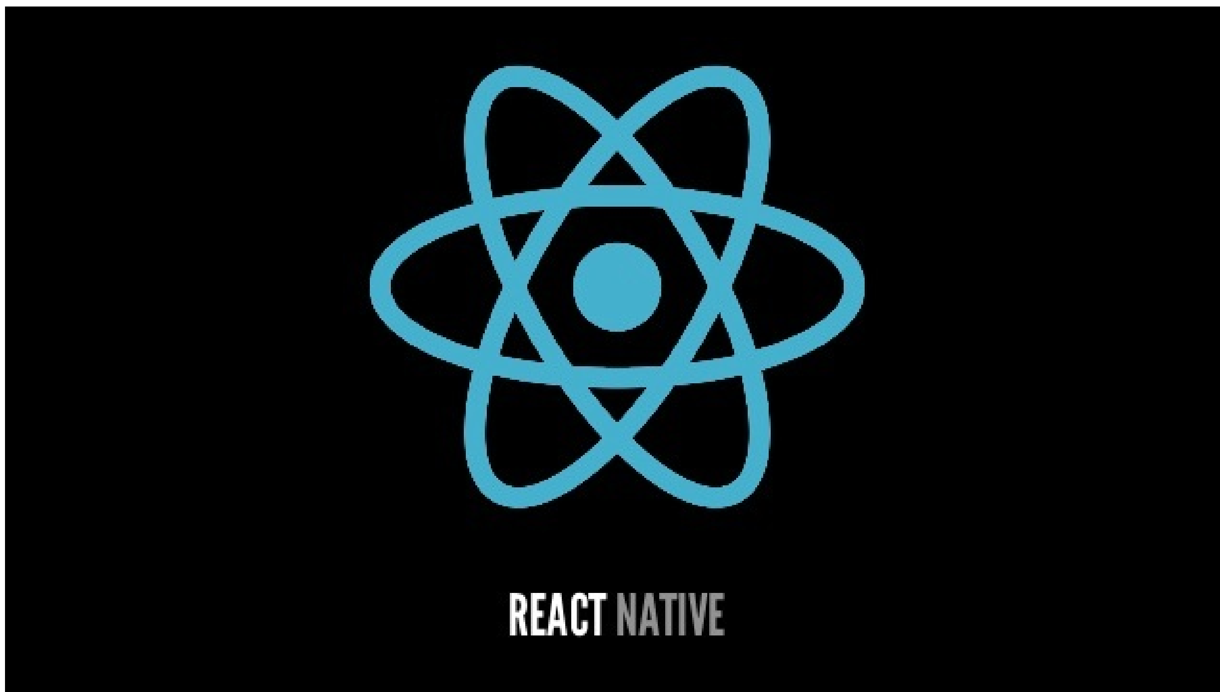
1. React/React Native 的ES5 ES6写法对照表
2. React on ES6+
3. react native 中es6语法解析
4. Learn ES2015
5. ECMAScript 6入门
6. React官方网站
7. React INTRO TO REACT.JS
8. React.createClass versus extends React.Component
9. react-native-coding-style
10. Airbnb React/JSX Style Guide
11. ECMAScript 6入门

:door: 任意门

| 回首页 | 上一章：用 React + Redux + Node（Isomorphic JavaScript）开发食谱分享网站 | 下一章：用 React Native + Firebase 开发跨平台行动应用程序 |

| 勘误、提问或许愿 |

附录二、用 React Native + Firebase 开发跨平台行动应用程序



前言

跨平台（ `Wirte once, Run Everywhere` ）一直以来是软体工程的万金油。过去一段时间市场上有许多尝试跨平台开发原生手机应用（Native Mobile App）的解决方案，尝试运用 HTML、CSS 和 JavaScript 等网页前端技术达到跨平台的效果，例如：运用 jQuery Mobile、Ionic 和 Framework7 等 Mobile UI 框架（Framework）结合 JavaScript 框架并搭配 Cordova/PhoneGap 进行跨平台手机应用开发。然而，因为这些解决方案通常都是运行在 `WebView` 之上，导致性能和体验要真正趋近于原生应用程序（Native App）还有一段路要走。

但是，随著 Facebook 工程团队开发的 React Native 横空出世，想尝试跨平台解决方案的开发者又有了新的选择。

React Native 特色

在正式开始开发 React Native App 之前我们先来介绍一下 React Native 的主要特色：

1. 使用 JavaScript（ES6+）和 React 打造跨平台原生应用程序（Learn once, write anywhere）
2. 使用 Native Components，更贴近原生使用者体验
3. 在 JavaScript 和 Native 之间的操作为非同步（Asynchronous）执行，并可用 Chrome 开发者工具除错，支持 `Hot Reloading`
4. 使用 Flexbox 进行排版和布局
5. 良好的可扩展性（Extensibility），容易整合 Web 生态系标准（XMLHttpRequest、navigator.geolocation 等）或是原生的组件或函数库（Objective-C、Java 或 Swift）
6. Facebook 已使用 React Native 于自家 Production App 且将持续维护，另外也有持续蓬勃发展的技术社群
7. 让 Web 开发者可以使用熟悉的技术切入 Native App 开发
8. 2015/3 发布 iOS 版本，2015/9 发布 Android 版本
9. 目前更新速度快，平均每两周发布新的版本。社群也还持续在寻找最佳实践，关于版本进展可以参考这个文件
10. 支持的操作系统为 >= Android 4.1 (API 16) 和 >= iOS 7.0

React Native 初体验

在了解了 React Native 特色后，我们准备开始开发我们的 React Native 应用程序！由于我们的范例可以让程序跨平台共用，所以你可以使用 iOS 和 Android 平台运行。不过若是想在 iOS 平台开发需要先准备 Mac OS 和安装 Xcode 开发工具，若是你准备使用 Android 平台的话建议先行安装 Android Studio 和 Genymotion 模拟器。在我们范例我们使用笔者使用的 MacO OS 操作系统并使用 Android 平台为主要范例，若有其他操作系统需求的读者可以参考 官方安装说明。

一开始请先安装 Node、Watchman 和 React Native command line 工具：

```
// 若你使用 Mac OS 你可以使用官网安装方式或是使用 homebrew 安装
$ brew install node
// watchman 可以监看文件是否有修改
$ brew install watchman
```

```
// 安装 React Native command line 工具
$ npm install -g react-native-cli
```

由于我们是要开发 Android 平台，所以必须安装：

1. 安装 JDK
2. 安装 Android SDK
3. 设定一些环境变量

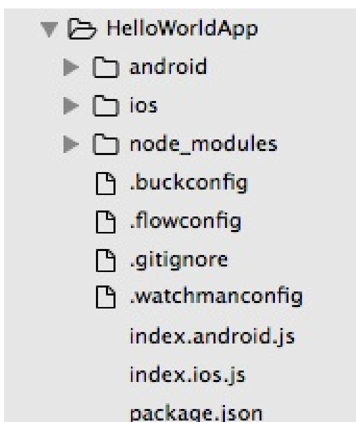
以上可以通过 Install Android Studio 官网和 官方安装说明 步骤完成。

现在，我们先通过一个简单的 `HelloWorldApp`，让大家感受一下 React Native 项目如何开发。

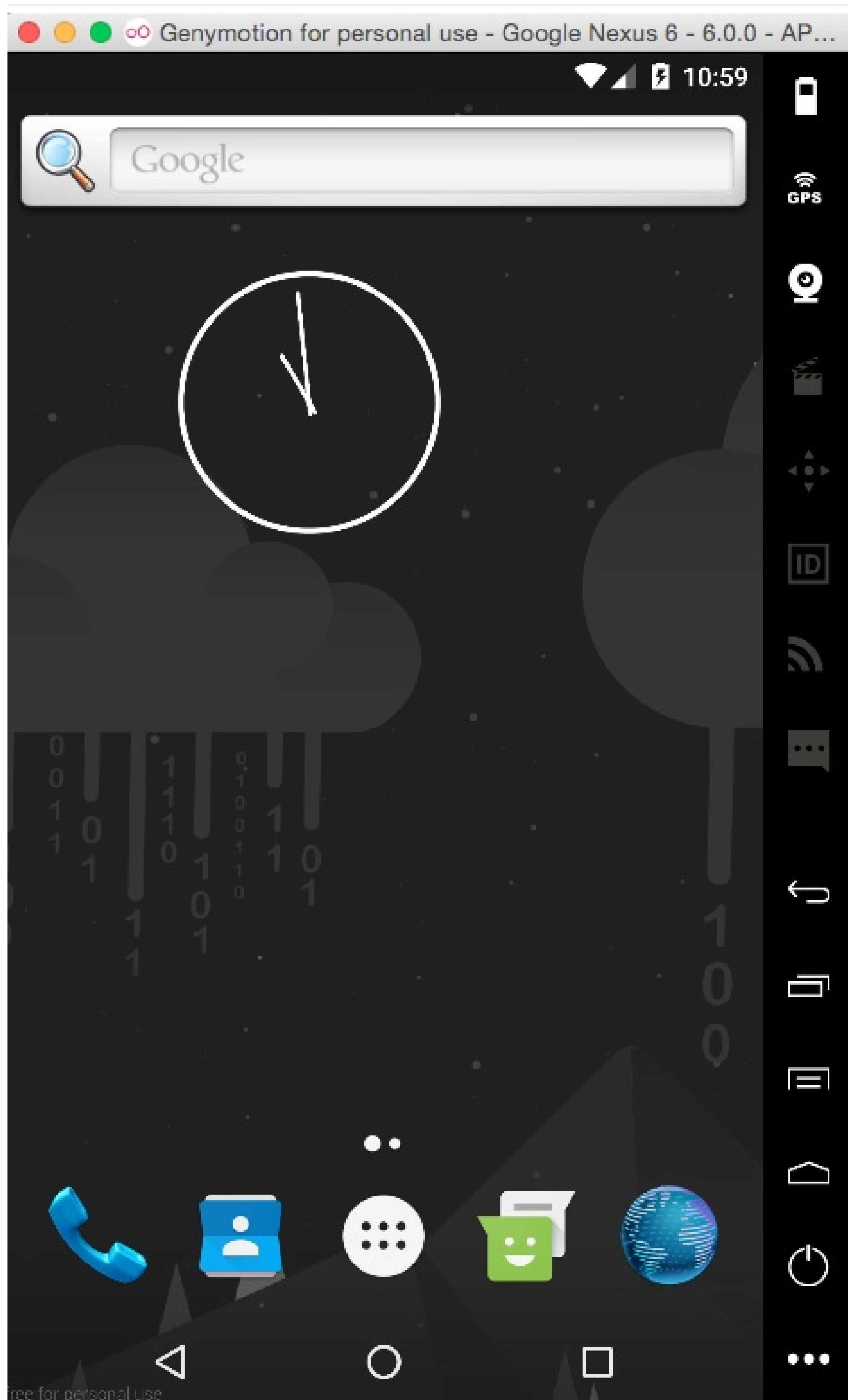
首先，我们先初始化一个 React Native Project：

```
$ react-native init HelloWorldApp
```

初始的文件夹结构：



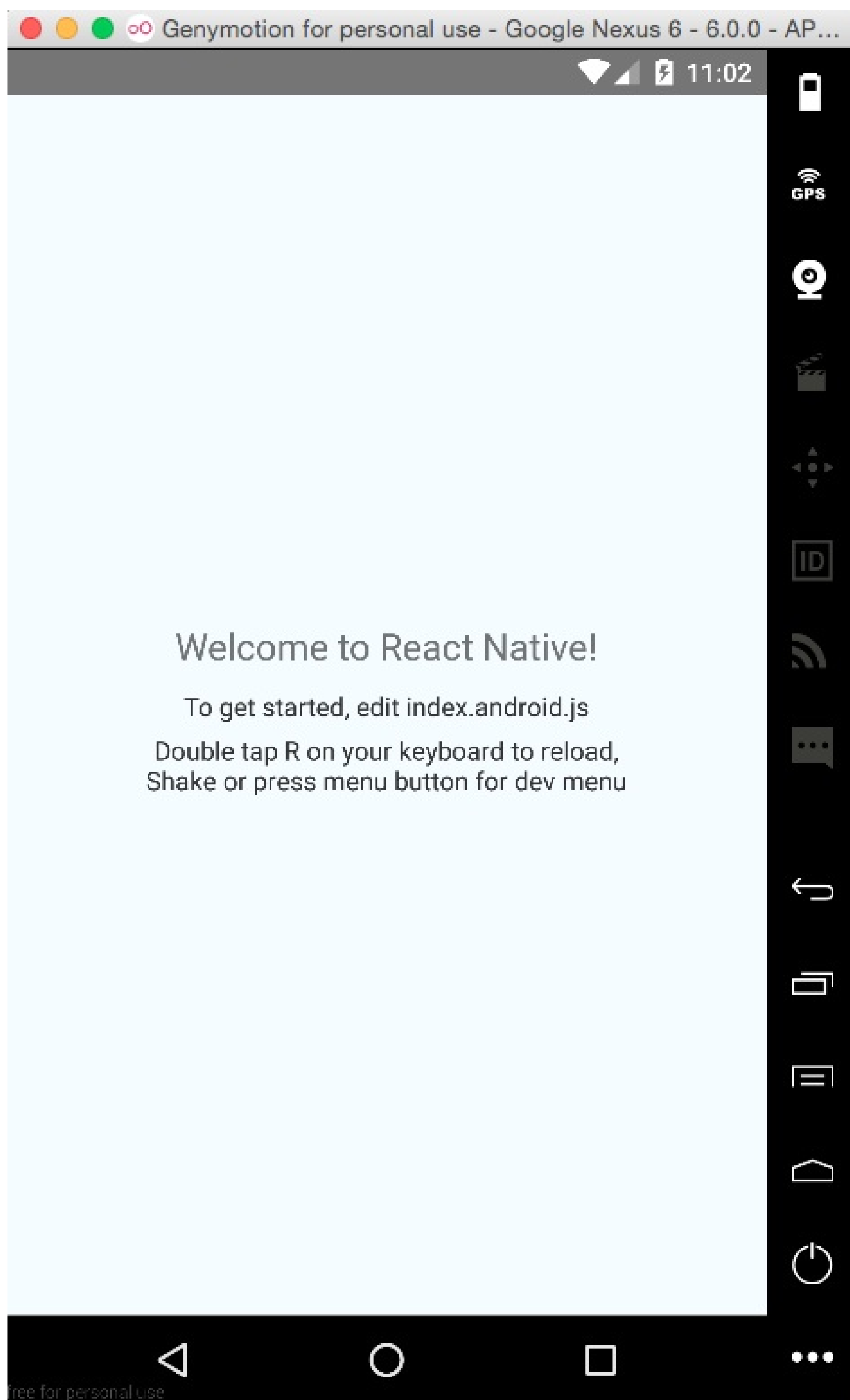
接下来请先安装注册 Genymotion，Genymotion 是一个通过电脑模拟 Android 系统的好用开发模拟器环境。安装完后可以打开并选择需要使用的屏幕大小和 API 版本的 Android 系统。部署好开发环境后就可以启动我们的服务：



若你是使用 Mac OS 操作系统的话可以执行 `run-ios` ，若是使用 Android 平台则使用 `run-android` 启动你的 App。在这边我们先使用 Android 平台进行开发（若你希望实机测试，请将电脑接上你的 Android 手机，记得确保 menu 中的 ip 地址要和电脑网络 相同。若是遇到连不到程序 server 且手机为 Android 5.0+ 系统，可以执行 `adb reverse tcp:8081 tcp:8081` ，详细情形可以参考官网说明）：

```
$ react-native run-android
```

如果一切顺利的话就可以在模拟器中看到初始画面：



接著打开 `index.android.js` 就可以看到以下代码：

```
import React, { Component } from 'react';
import {
  AppRegistry,
  StyleSheet,
```

```
  Text,
  View
} from 'react-native';

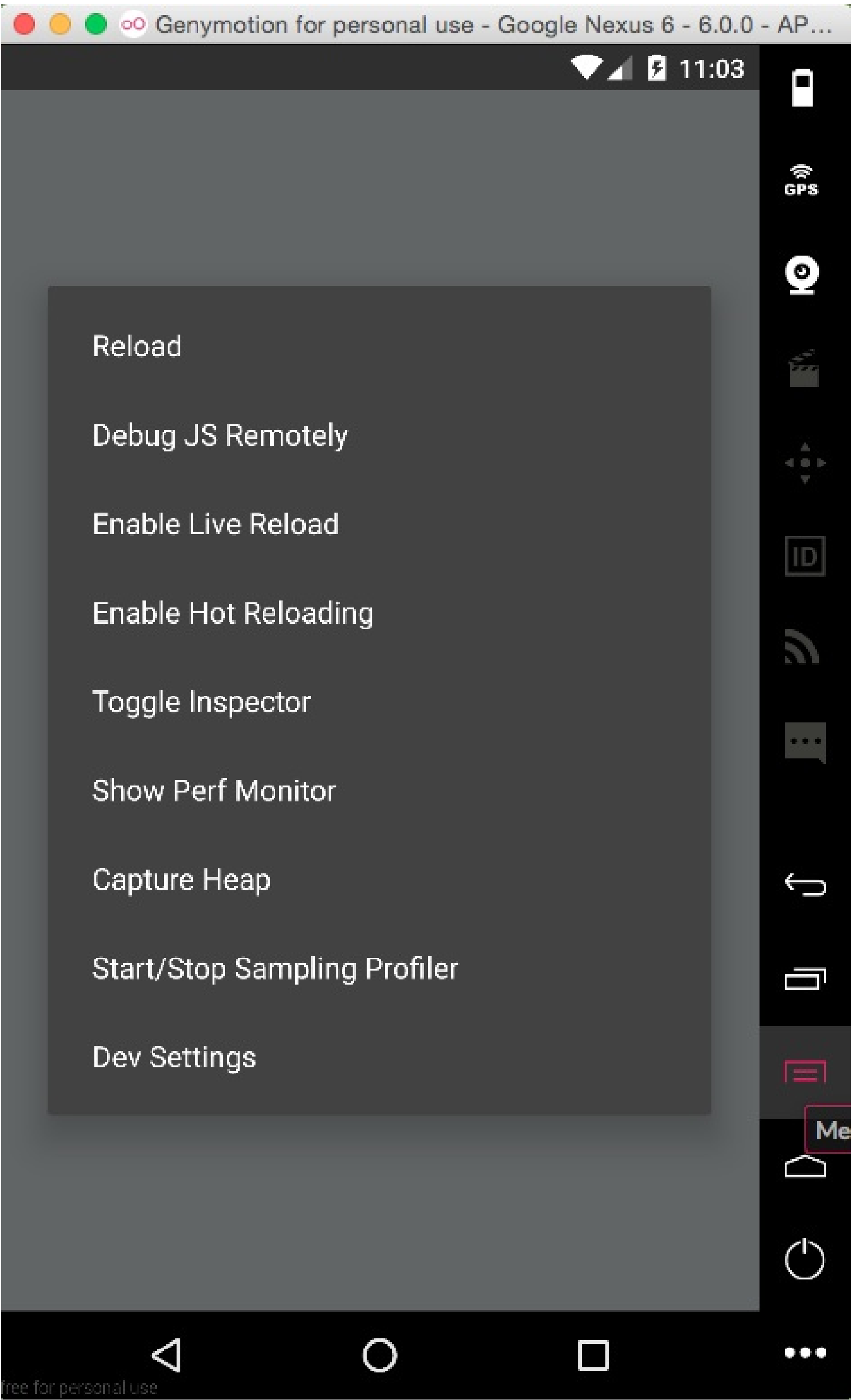
// 组件式的开发方式和 React 如出一辙，但要注意的是在 React Native 中我们不使用 HTML 元素而是使用 React Native 组件进行开发，这也符合 Learn once, write anywhere 的原则。
class HelloWorldApp extends Component {
  render() {
    return (
      <View style={styles.container}>
        <Text style={styles.welcome}>
          Welcome to React Native!
        </Text>
        <Text style={styles.instructions}>
          To get started, edit index.android.js
        </Text>
        <Text style={styles.instructions}>
          Double tap R on your keyboard to reload,{'\n'}
          Shake or press menu button for dev menu
        </Text>
      </View>
    );
  }
}

// 在 React Native 中 styles 是使用 JavaScript 形式来撰写，与一般 CSS 比较不同的是他使用驼峰式的属性命名：
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
    backgroundColor: '#F5FCFF',
  },
  welcome: {
    fontSize: 20,
    textAlign: 'center',
    margin: 10,
  },
```

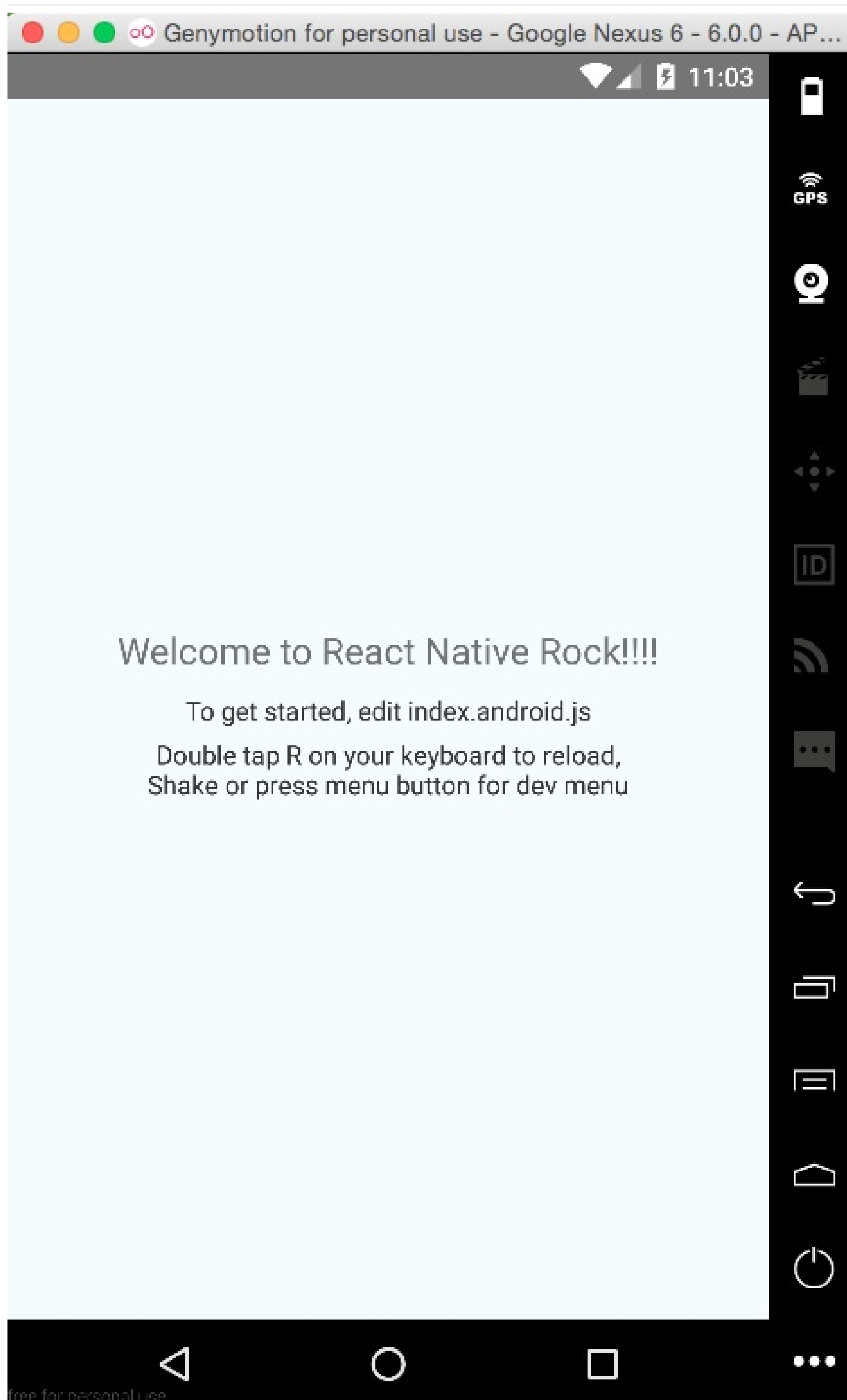
```
  instructions: {
    textAlign: 'center',
    color: '#333333',
    marginBottom: 5,
  },
});

// 告诉 React Native App 你的进入点：
AppRegistry.registerComponent('HelloWorldApp', () => HelloWorldApp);
```

由于 React Native 有支持 `Hot Reloading` ，若我们更改了程序内容，我们可以使用打开模拟器 Menu 重新刷新页面，此时就可以在看到原本的 `Welcome to React Native!` 文字已经改成 `Welcome to React Native Rock!!!!`



Genymotion for personal use - Google Nexus 6 - 6.0.0 - AP...
11:03
GPS
ID
Reload
Debug JS Remotely
Enable Live Reload
Enable Hot Reloading
Toggle Inspector
Show Perf Monitor
Capture Heap
Start/Stop Sampling Profiler
Dev Settings
Me
free for personal use



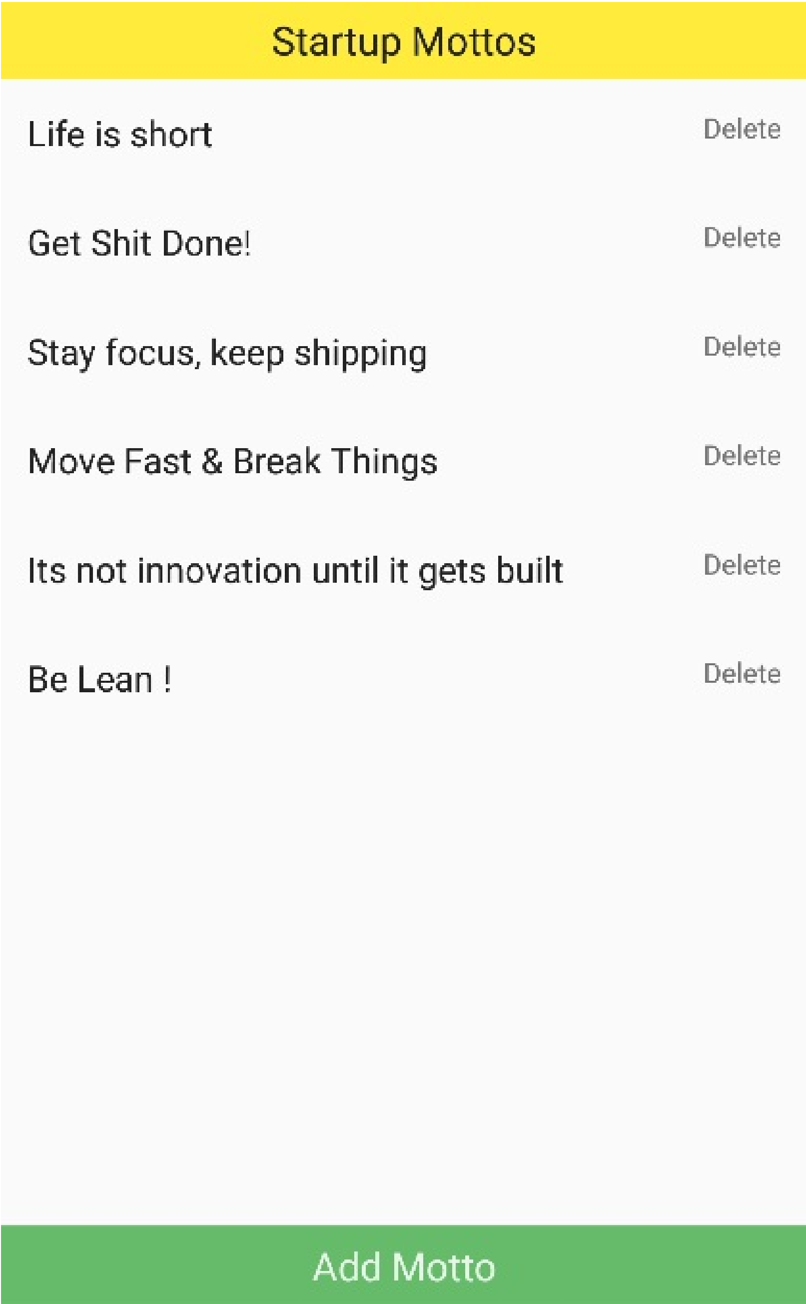
嗯，有没有感觉在开发网页的感觉？

动手实践

相信看到这里读者们一定等不及想大展身手，使用 React Native 开发你第一个 App。俗话说学习一项新技术最好的方式就是做一个 TodoApp。所以，接下来的文章，笔者将带大家使用 React Native 结合 Redux/ImmutableJS 和 Firebase 开发一

个记录和删除名言佳句（Mottos）的 Mobile App！

项目成果截图



Please Keyin your Motto!

Be Lean !

Cancel Submit

环境安装与设定

相关包安装：

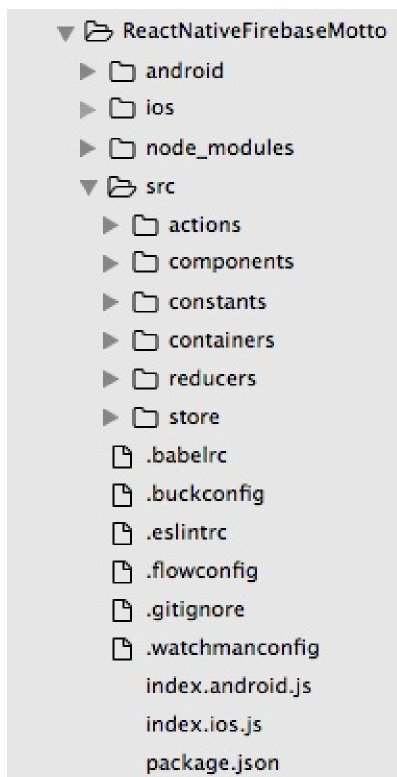
```
$ npm install --save redux react-redux immutable redux-immutable
 redux-actions uuid firebase
```

```
$ npm install --save-dev babel-core babel-eslint babel-loader ba
bel-preset-es2015 babel-preset-react babel-preset-react-native e
slint-plugin-react-native  eslint eslint-config-airbnb eslint-lo
ader eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-r
eact redux-logger
```

安装完相关工具后我们可以初始化我们项目：

```
// 注意项目不能使用 - 或 _ 命名
$ react-native init ReactNativeFirebaseMotto
$ cd ReactNativeFirebaseMotto
```

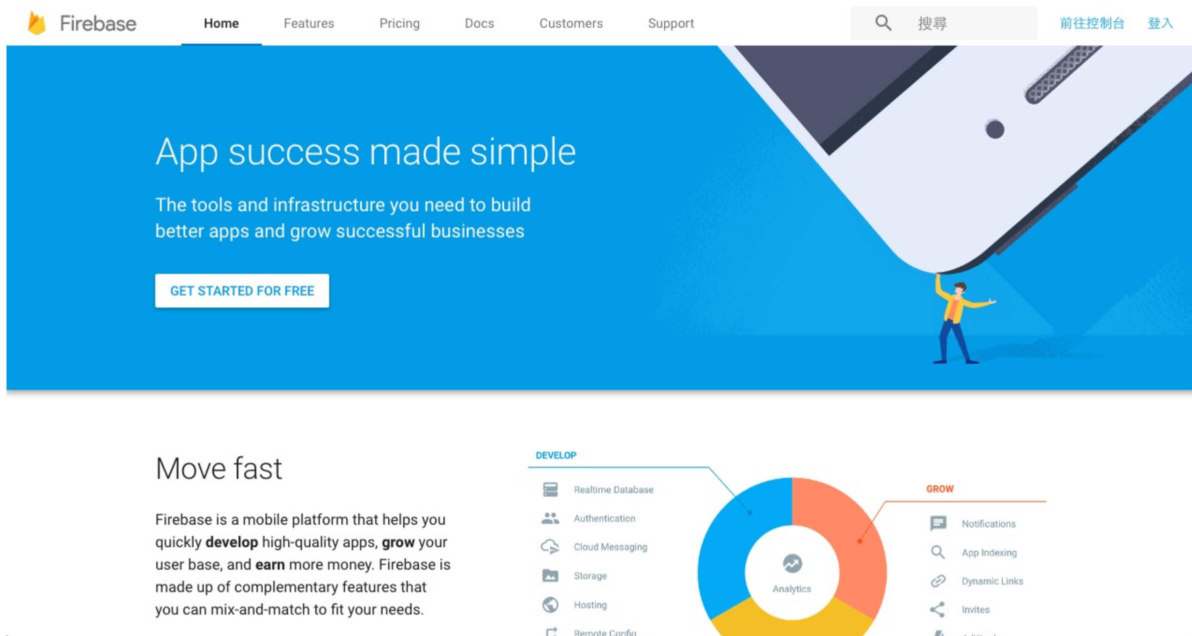
我们先准备一下我们文件夹架构，将它设计成：



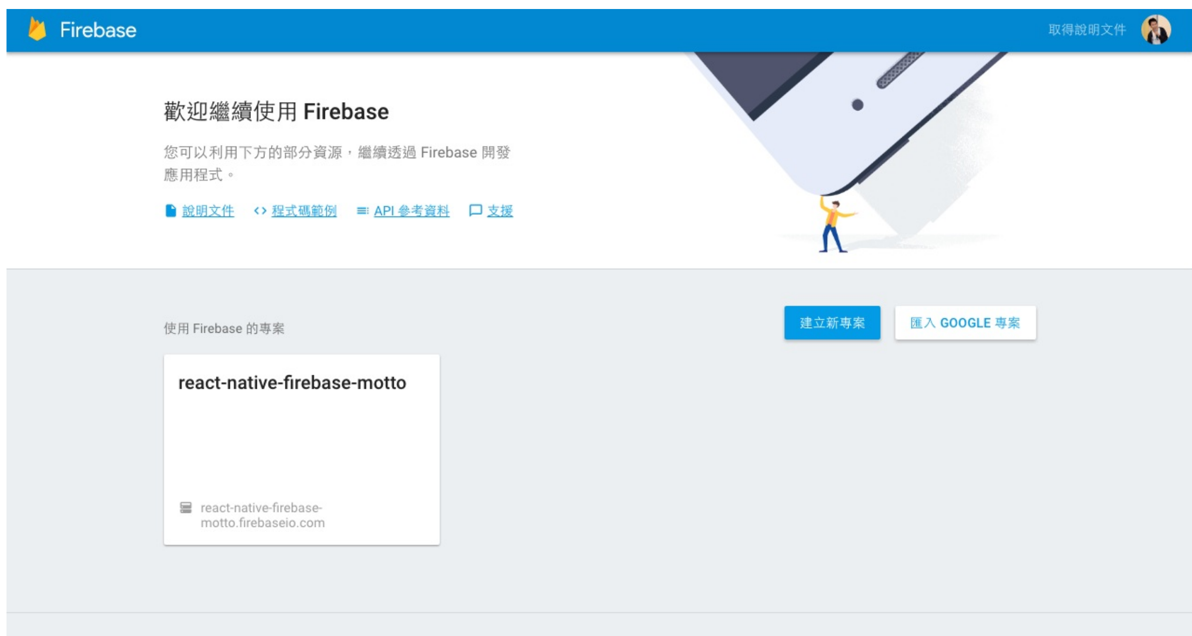
Firebase 简介与设定

在这个项目中我们会使用到 Firebase 这个 `Back-End as Service` 的服务，也就是说我们不用自己建立后端程序数据库，只要使用 Firebase 所提供的 API 就好像有了一个 NoSQL 数据库一样，当然 Firebase 不单只有提供数据储存的功能，但限于篇幅我们这边将只介绍数据储存的功能。

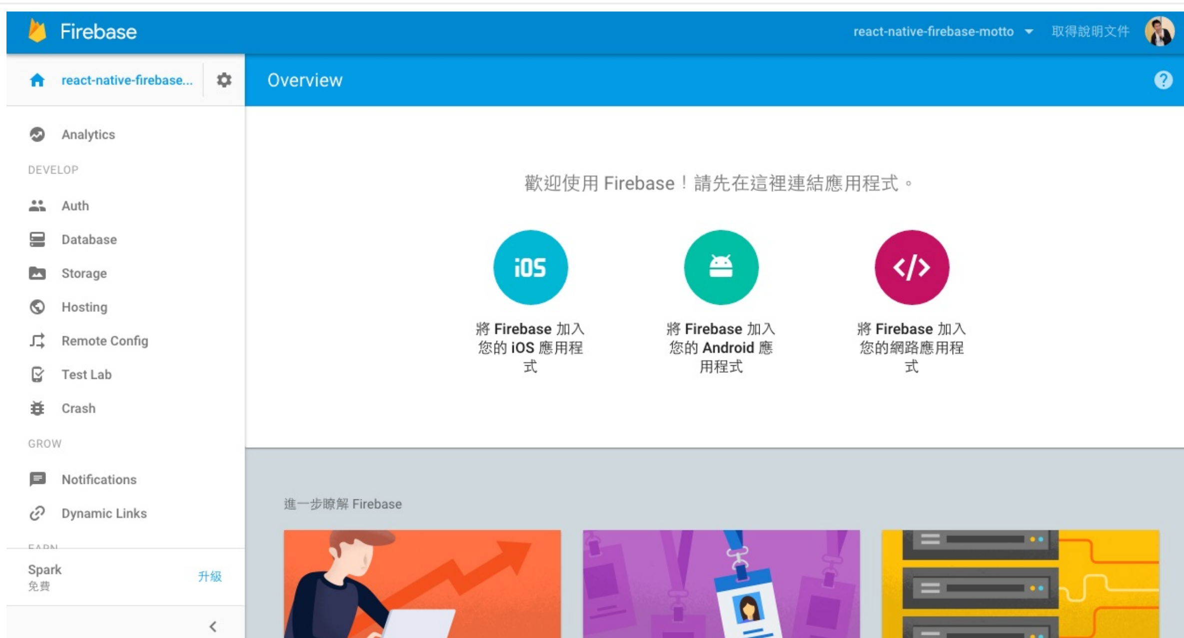
1. 首先我们进到 Firebase 首页



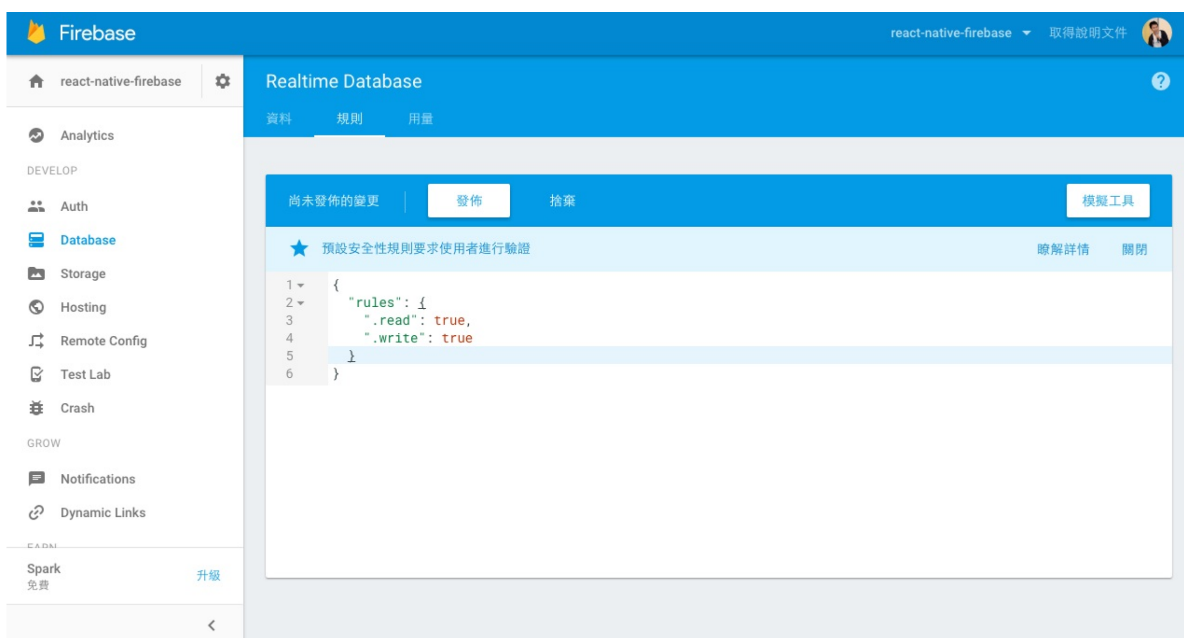
2. 登入后点选建立项目，依照自己想取的项目名称命名



3. 选择将 Firebase 加入你的网络应用程序的按钮可以取得 App ID 的 config 数据，待会我们将会使用到



4. 点选左边选单中的 Database 并点选 Realtime Database Tab 中的规则



设定改为，在范例中为求简单，我们先不用验证方式即可操作：

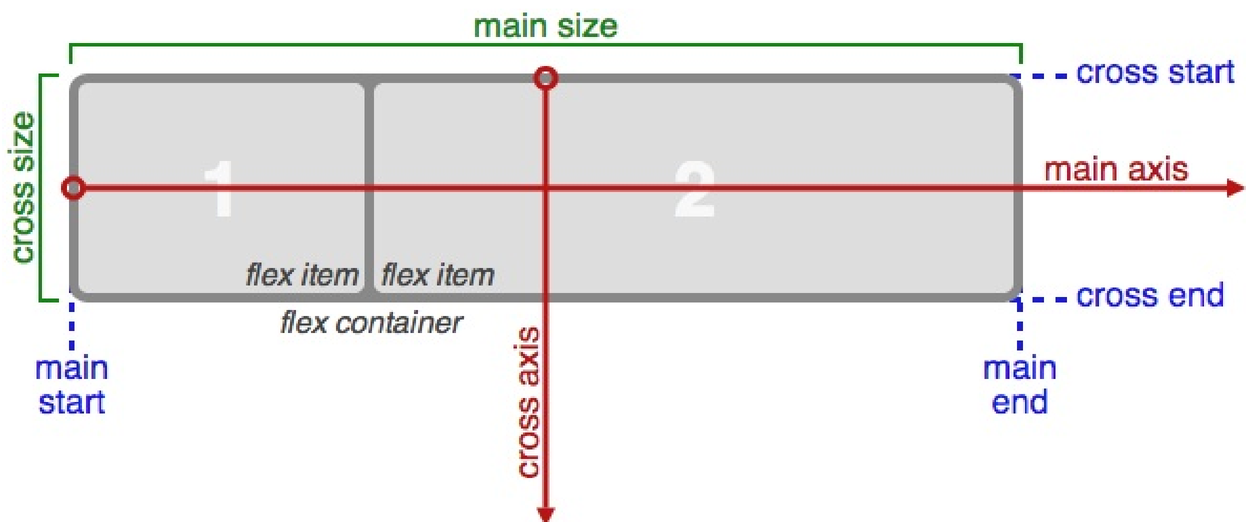
```
{
 "rules": {
   ".read": true,
   ".write": true
 }
}
```

Firebase 在使用上有许多优点，其中一个使用 Back-End As Service 的好处是你可以专注在应用程序的开发便免花过多时间处理后端基础建设的部份，更可以让 Back-End 共用在不同的 client side 中。此外 Firebase 在和 React 整合上也十分容易，你可以想成 Firebase 负责数据的储存，通过 API 和 React 组件互动，Redux 负责接收管理 client state，若是监听到 Firebase 后端数据更新后同步更新 state 并重新 render 页面。

使用 Flexbox 进行 UI 布局设计

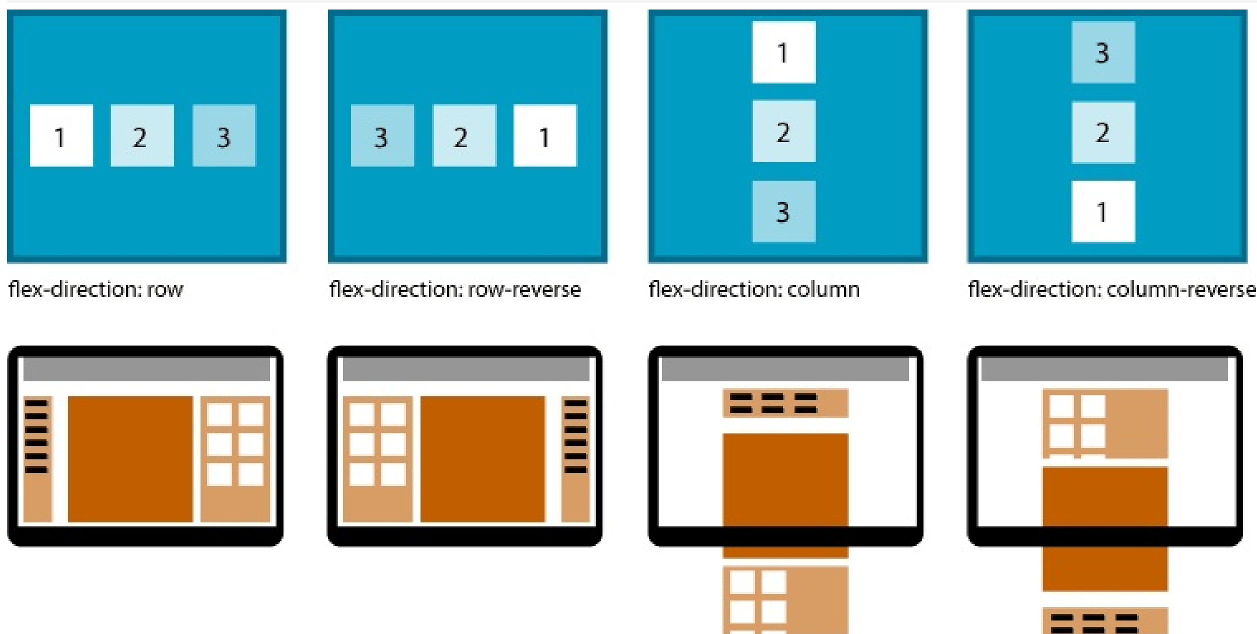
在 React Native 中是使用 `Flexbox` 进行排版，若读者对于 Flexbox 尚不熟悉，建议可以参考这篇文章，若有需要游戏化的学习工具，也非常推荐这两个教学小游戏：FlexDefense、FLEXBOX FROGGY。

事实上我们可以将 Flexbox 视为一个箱子，最外层是 `flex containers` 、内层包的是 `flex items`，在属性上也有分是针对 `flex containers` 还是针对是 `flex items` 设计的。在方向性上由左而右是 `main axis`，而上到下是 `cross axis`。



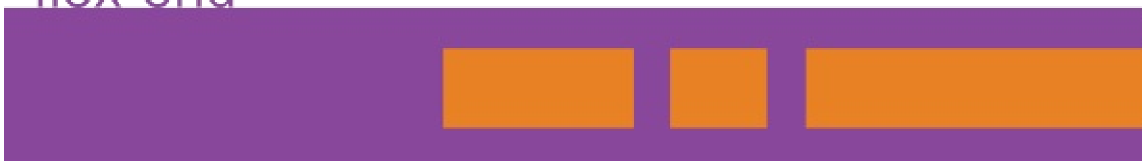
在 Flexbox 有许多属性值，其中最重要的当数 `justifyContent` 和 `alignItems` 以及 `flexDirection`（注意 React Native Style 都是驼峰式写法），所以我们这边主要介绍这三个属性：

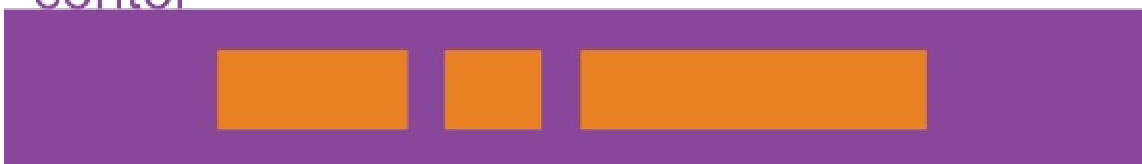
Flex Direction 负责决定整个 `flex containers` 的方向，预设为 `row` 也可以改为 `column` 、 `row-reverse` 和 `column-reverse` 。



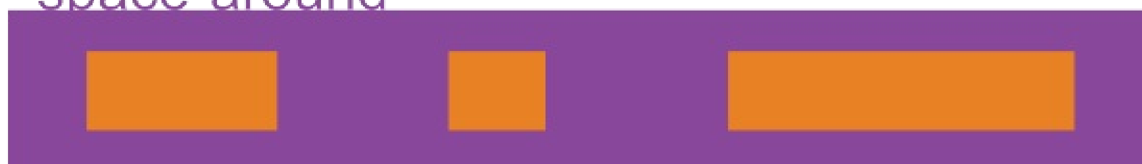
Justify Content 负责决定整个 `flex containers` 内的 items 的水平摆设，主要属性值有：`flex-start`、`flex-end`、`center`、`space-between`、`space-around`。



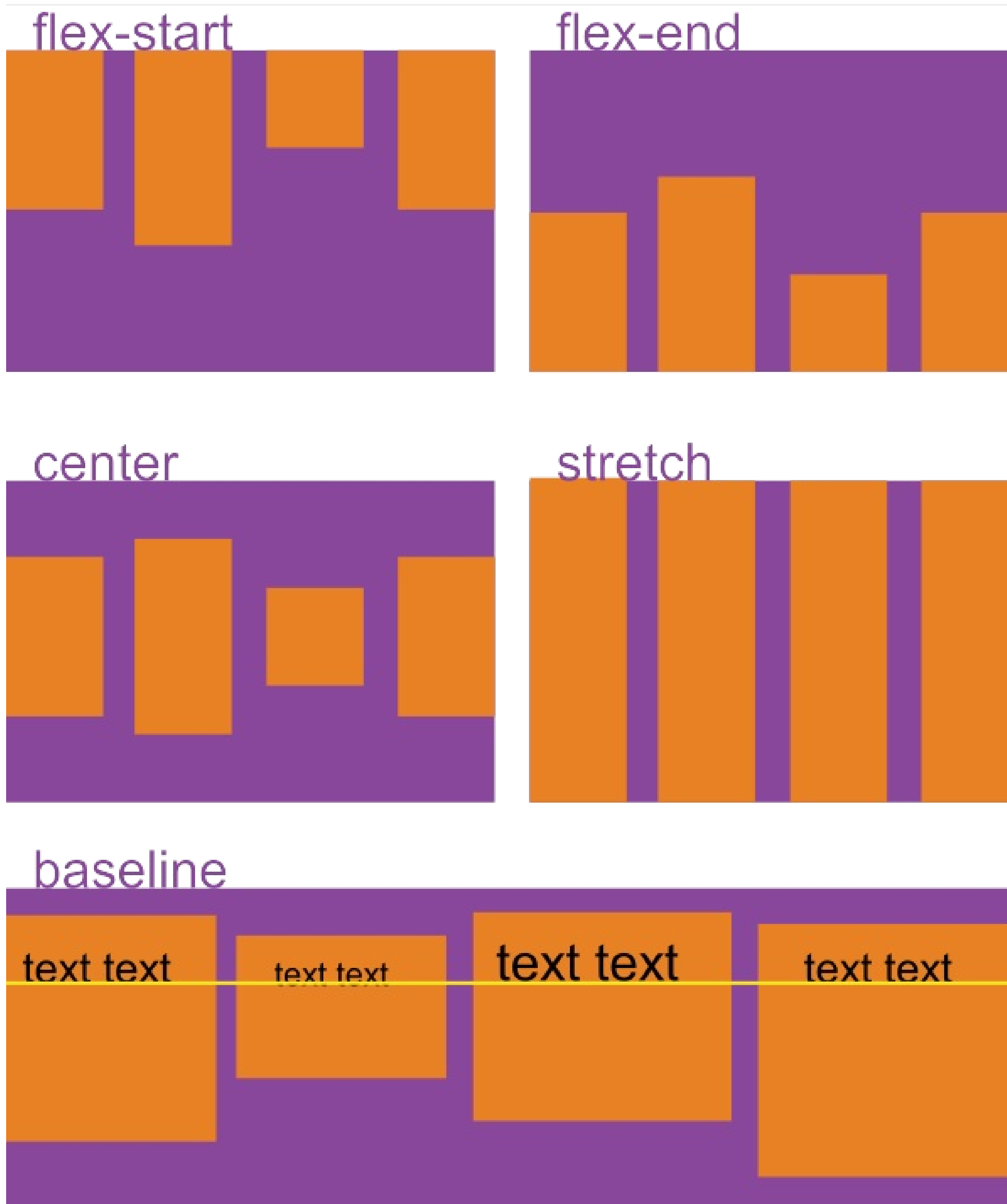








Align Items 负责决定整个 `flex containers` 内的 items 的垂直摆设，主要属性值有：`flex-start`、`flex-end`、`center`、`stretch`、`baseline`。



动手实践

有了前面的准备，现在我们终于要开始进入核心的应用程序开发了！

首先我们先设定好整个 App 的入口文件 `index.android.js` ，在这个文件中我们设定了初始化的设定和主要组件 `<Main />` ：

```
/**
 * Sample React Native App
 * https://github.com/facebook/react-native
 * @flow
 */

import React, { Component } from 'react';
import {
  AppRegistry,
  Text,
  View
} from 'react-native';
import Main from './src/components/Main';

class ReactNativeFirebaseMotto extends Component {
  render() {
    return (
      <Main />
    );
  }
}

AppRegistry.registerComponent('ReactNativeFirebaseMotto', () =>
ReactNativeFirebaseMotto);
```

在 `src/components/Main/Main.js` 中我们设定好整个 Component 的布局和并将 `Firebase` 引入并初始化，将操作 Firebase 数据库的参考往下传，根节点我们命名为 `items`，所以之后所有新增的 motto 都会在这个根节点之下并拥有特定的 key 值。在 Main 我们同样规划了整个布局，包括：`<ToolBar />`、`<MottoListContainer />`、`<ActionButtonContainer />`、`<InputModalContainer />`。

```
import React from 'react';
import ReactNative from 'react-native';
import { Provider } from 'react-redux';
import ToolBar from '../ToolBar';
import MottoListContainer from '../../containers/MottoListContai
ner';
import ActionButtonContainer from '../../containers/ActionButton
Container';
import InputModalContainer from '../../containers/InputModalCont
ainer';
import ListItem from '../ListItem';
import * as firebase from 'firebase';
// 将 Firebase 的 config 值引入
import { firebaseConfig } from '../../constants/config';
// 引用 Redux store
import store from '../../store';
const { View, Text } = ReactNative;

// Initialize Firebase
const firebaseApp = firebase.initializeApp(firebaseConfig);
// Create a reference with .ref() instead of new Firebase(url)
const rootRef = firebaseApp.database().ref();
const itemsRef = rootRef.child('items');

// 将 Redux 的 store 通过 Provider 往下传
const Main = () => (
  <Provider store={store}>
    <View>
      <ToolBar style={styles.toolBar} />
      <MottoListContainer itemsRef={itemsRef} />
      <ActionButtonContainer />
      <InputModalContainer itemsRef={itemsRef} />
    </View>
  </Provider>
);

export default Main;
```

设定完了基本的布局方式后我们来设定 Actions 和其使用的常数， `src/actions/mottoActions.js` ：

```
export const GET_MOTTOS = 'GET_MOTTOS';
export const CREATE_MOTTO = 'CREATE_MOTTO';
export const SET_IN_MOTTO = 'SET_IN_MOTTO';
export const TOGGLE_MODAL = 'TOGGLE_MODAL';
```

我们在 constants 文件夹中也设定了我们整个 data 的数据结构，以下是 `src/constants/models.js` ：

```
import Immutable from 'immutable';

export const MottoState = Immutable.fromJS({
  mottos: [],
  motto: {
    id : '',
    text: '',
    updatedAt: '',
  }
});

export const UiState = Immutable.fromJS({
  isModalVisible: false,
});
```

还记得我们提到的 Firebase config 吗？这边我们把相关的设定档放在 `src/configs/config.js` 中：

```
export const firebaseConfig = {
  apiKey: "apiKey",
  authDomain: "authDomain",
  databaseURL: "databaseURL",
  storageBucket: "storageBucket",
};
```

在我们应用程序中同样使用了 `redux` 和 `redux-actions` 。在这个范例中我们设计了：GET_MOTTOS、CREATE_MOTTO、SET_IN_MOTTO 三个操作 motto 的 action，分别代表从 Firebase 取出数据、新增数据和 set 数据。以下是 `src/actions/mottoActions.js` ：

```
import { createAction } from 'redux-actions';
import {
  GET_MOTTOS,
  CREATE_MOTTO,
  SET_IN_MOTTO,
} from '../constants/actionTypes';

export const getMottos = createAction('GET_MOTTOS');
export const createMotto = createAction('CREATE_MOTTO');
export const setInMotto = createAction('SET_IN_MOTTO');
```

同样地，由于我们设计了当使用者想新增 motto 时会跳出 modal，所以我们可以设定一个 `TOGGLE_MODAL` 负责开关 modal 的 state。以下是 `src/actions/uiActions.js` ：

```
import { createAction } from 'redux-actions';
import {
  TOGGLE_MODAL,
} from '../constants/actionTypes';

export const toggleModal = createAction('TOGGLE_MODAL');
```

以下是 `src/actions/index.js` ，用来输出我们的 actions：

```
export * from './uiActions';
export * from './mottoActions';
```

设定完我们的 actions 后我们来设定 reducers，在这边我们同样使用 `redux-actions` 整合 `ImmutableJS` ，

```
import { handleActions } from 'redux-actions';
// 引入 initialState
import {
  MottoState
} from '../../constants/models';

import {
  GET_MOTTOS,
  CREATE_MOTTO,
  SET_IN_MOTTO,
} from '../../constants/actionTypes';

// 通过 set 和 seIn 可以产生 newState
const mottoReducers = handleActions({
  GET_MOTTOS: (state, { payload }) => (
    state.set(
      'mottos',
      payload.mottos
    )
  ),
  CREATE_MOTTO: (state) => (
    state.set(
      'mottos',
      state.get('mottos').push(state.get('motto'))
    )
  ),
  SET_IN_MOTTO: (state, { payload }) => (
    state.setIn(
      payload.path,
      payload.value
    )
  )
}, MottoState);

export default mottoReducers;
```

以下是 `src/reducers/uiState.js` ：

```
import { handleActions } from 'redux-actions';
import {
  UiState,
} from '../../constants/models';

import {
  TOGGLE_MODAL,
} from '../../constants/actionTypes';

// modal 的显示与否
const uiReducers = handleActions({
  TOGGLE_MODAL: (state) => (
    state.set(
      'isModalVisible',
      !state.get('isModalVisible')
    )
  ),
}, UiState);

export default uiReducers;
```

以下是 `src/reducers/index.js` ，将所有 reducers combine 在一起：

```
import { combineReducers } from 'redux-immutable';
import ui from './ui/uiReducers';
import motto from './data/mottoReducers';

const rootReducer = combineReducers({
  ui,
  motto,
});

export default rootReducer;
```

通过 `src/store/configureStore.js` 将 reducers 和 initialState 以及要使用的 middleware 整合成 store：

```
import { createStore, applyMiddleware } from 'redux';
import createLogger from 'redux-logger';
import Immutable from 'immutable';
import rootReducer from '../reducers';

const initialState = Immutable.Map();

export default createStore(
  rootReducer,
  initialState,
  applyMiddleware(createLogger({ stateTransformer: state => stat
e.toJS() }))
);
```

设定完数据层的架构后，我们又重新回到 View 的部份，我们开始依序设定我们的 Component 和 Container。首先，我们先设计我们的标题列 ToolBar，以下是 `src/components/ToolBar/ToolBar.js` ：

```
import React from 'react';
import ReactNative from 'react-native';
import styles from './toolBarStyles';
const { View, Text } = ReactNative;

const ToolBar = () => (
  <View style={styles.toolBarContainer}>
    <Text style={styles.toolBarText}>Startup Mottos</Text>
  </View>
);

export default ToolBar;
```

以下是 `src/components/ToolBar/toolBarStyles.js` ，将底色设定为黄色，文字置中：

```
import { StyleSheet } from 'react-native';

export default StyleSheet.create({
  toolBarContainer: {
    height: 40,
    justifyContent: 'center',
    alignItems: 'center',
    flexDirection: 'column',
    backgroundColor: '#ffeb3b',
  },
  toolBarText: {
    fontSize: 20,
    color: '#212121'
  }
});
```

以下是 `src/components/MottoList/MottoList.js` ，这个 Component 中稍微复杂一些，主要是使用到了 React Native 中的 ListView Component 将数据结构传进 dataSource，通过 renderRow 把一个个 row 给 render 出来，过程中我们通过 `!Immutable.is(r1.get('id'), r2.get('id'))` 去判断整个 ListView 画面是否需要 loading 新的 item 进来，这样就可以提高整个 ListView 的性能。

```
import React, { Component } from 'react';
import ReactNative from 'react-native';
import Immutable from 'immutable';
import ListItem from '../ListItem';
import styles from './mottoStyles';
const { View, Text, ListView } = ReactNative;

class MottoList extends Component {
  constructor(props) {
    super(props);
    this.renderListItem = this.renderListItem.bind(this);
    this.listenForItems = this.listenForItems.bind(this);
    this.ds = new ListView.DataSource({
      rowHasChanged: (r1, r2) => !Immutable.is(r1.get('id'), r2.
get('id')),
    })
```

```
  }
  renderListItem(item) {
    return (
      <ListItem item={item} onDeleteMotto={this.props.onDeleteMo
tto} itemsRef={this.props.itemsRef} />
    );
  }
  listenForItems(itemsRef) {
    itemsRef.on('value', (snap) => {
      if(snap.val() === null) {
        this.props.onGetMottos(Immutable.fromJS([]));
      } else {
        this.props.onGetMottos(Immutable.fromJS(snap.val()));
      }
    });
  }
  componentDidMount() {
    this.listenForItems(this.props.itemsRef);
  }
  render() {
    return (
      <View>
        <ListView
          style={styles.listView}
          dataSource={this.ds.cloneWithRows(this.props.mottos.to
Array())}
          renderRow={this.renderListItem}
          enableEmptySections={true}
        />
      </View>
    );
  }
}

export default MottoList;
```

以下是 `src/components/MottoList/mottoListStyles.js` ，我们使用到了 Dimensions，可以根据屏幕的高度来设定整个 ListView 高度：

```
import { StyleSheet, Dimensions } from 'react-native';
const { height } = Dimensions.get('window');
export default StyleSheet.create({
  listView: {
    flex: 1,
    flexDirection: 'column',
    height: height - 105,
  },
});
```

以下是 `src/components/ListItem/ListItem.js` ，我们从 props 收到了上层传进来的 motto item，显示出 motto 文字内容。当我们点击 `<TouchableHighlight>` 时就会删除该 motto。

```
import React from 'react';
import ReactNative from 'react-native';
import styles from './listItemStyles';
const { View, Text, TouchableHighlight } = ReactNative;

const ListItem = (props) => {
  return (
    <View style={styles.listItemContainer}>
      <Text style={styles.listItemText}>{props.item.get('text')}
</Text>
      <TouchableHighlight onPress={props.onDeleteMotto(props.ite
m.get('id'), props.itemsRef)}>
        <Text>Delete</Text>
      </TouchableHighlight>
    </View>
  )
};

export default ListItem;
```

以下是 `src/components/ListItem/listItemStyles.js` ：

```
import { StyleSheet } from 'react-native';

export default StyleSheet.create({
  listItemContainer: {
    flex: 1,
    flexDirection: 'row',
    padding: 10,
    margin: 5,
  },
  listItemText: {
    flex: 10,
    fontSize: 18,
    color: '#212121',
  }
});
```

以下是 `src/components/ActionButton/ActionButton.js` ，当点击了按钮则会触发 onToggleModal 方法，出现新增 motto 的 modal：

```
import React from 'react';
import ReactNative from 'react-native';
import styles from './actionButtonStyles';
const { View, Text, Modal, TextInput, TouchableHighlight } = ReactNative;

const ActionButton = (props) => (
  <TouchableHighlight onPress={props.onToggleModal}>
    <View style={styles.buttonContainer}>
        <Text style={styles.buttonText}>Add Motto</Text>
    </View>
  </TouchableHighlight>
);

export default ActionButton;
```

以下是 `src/components/ActionButton/actionButtonStyles.js` ：

```
import { StyleSheet } from 'react-native';

export default StyleSheet.create({
  buttonContainer: {
    height: 40,
    justifyContent: 'center',
    alignItems: 'center',
    flexDirection: 'column',
    backgroundColor: '#66bb6a',
  },
  buttonText: {
    fontSize: 20,
    color: '#e8f5e9'
  }
});
```

以下是 `src/components/InputModal/InputModal.js` ，其主要负责 Modal Component 的设计，当输入内容会触发 onChangeMottoText 发出 action，注意的是当按下送出键，同时会把 Firebase 的参考 itemsRef 送入 onCreateMotto 中，方便通过 API 去即时新增到 Firebase Database，并更新 client state 和重新渲染了 View：

```
import React from 'react';
import ReactNative from 'react-native';
import styles from './inputModelStyles';
const { View, Text, Modal, TextInput, TouchableHighlight } = Rea
ctNative;
const InputModal = (props) => (
  <View>
    <Modal
      animationType={"slide"}
      transparent={false}
      visible={props.isModalVisible}
      onRequestClose={props.onToggleModal}
      >
     <View>
      <View>
        <Text style={styles.modalHeader}>Please Keyin your Motto!
```

```
</Text>
        <TextInput
          onChangeText={props.onChangeMottoText}
        />
        <View style={styles.buttonContainer}>
          <TouchableHighlight
            onPress={props.onToggleModal}
            style={[styles.cancelButton]}
          >
            <Text
              style={styles.buttonText}
            >
              Cancel
            </Text>
          </TouchableHighlight>
          <TouchableHighlight
            onPress={props.onCreateMotto(props.itemsRef)}
            style={[styles.submitButton]}
          >
            <Text
              style={styles.buttonText}
            >
              Submit
            </Text>
          </TouchableHighlight>
        </View>
      </View>
     </View>
    </Modal>
  </View>
);

export default InputModal;
```

以下是 `src/components/InputModal/inputModalStyles.js`：

```
import { StyleSheet } from 'react-native';

export default StyleSheet.create({
  modalHeader: {
    flex: 1,
    height: 30,
    padding: 10,
    flexDirection: 'row',
    backgroundColor: '#ffc107',
    fontSize: 20,
  },
  buttonContainer: {
    flex: 1,
    flexDirection: 'row',
  },
  button: {
    borderRadius: 5,
  },
  cancelButton: {
    flex: 1,
    height: 40,
    alignItems: 'center',
    justifyContent: 'center',
    backgroundColor: '#eceff1',
    margin: 5,
  },
  submitButton: {
    flex: 1,
    height: 40,
    alignItems: 'center',
    justifyContent: 'center',
    backgroundColor: '#4fc3f7',
    margin: 5,
  },
  buttonText: {
    fontSize: 20,
  }
});
```

设定完了 Component，我们来探讨一下 Container 的部份。以下是 `src/containers/ActionButtonContainer/ActionButtonContainer.js`：

```
import { connect } from 'react-redux';
import ActionButton from '../../components/ActionButton';
import {
  toggleModal,
} from '../../actions';

export default connect(
  (state) => ({}),
  (dispatch) => ({
    onToggleModal: () => (
      dispatch(toggleModal())
    )
  })
)(ActionButton);
```

以下是 `src/containers/InputModalContainer/InputModalContainer.js`：

```
import { connect } from 'react-redux';
import InputModal from '../../components/InputModal';
import Immutable from 'immutable';

import {
  toggleModal,
  setInMotto,
  createMotto,
} from '../../actions';
import uuid from 'uuid';

export default connect(
  (state) => ({
    isModalVisible: state.getIn(['ui', 'isModalVisible']),
    motto: state.getIn(['motto', 'motto']),
  }),
  (dispatch) => ({
    onToggleModal: () => (
      dispatch(toggleModal())
```

```
    ),
    onChangeMottoText: (text) => (
      dispatch(setInMotto({ path: ['motto', 'text'], value: text
 }))
    ),
    // 新增 motto 是通过 itemsRef 将新增的 motto push 进去，新增后要把
本地端的 motto 清空，并关闭 modal：
    onCreateMotto: (motto) => (itemsRef) => () => {
      itemsRef.push({ id: uuid.v4(), text: motto.get('text'), up
datedAt: Date.now() });
      dispatch(setInMotto({ path: ['motto'], value: Immutable.fr
omJS({ id: '', text: '', updatedAt: '' })}));
      dispatch(toggleModal());
    }
  }),
  (stateToProps, dispatchToProps, ownProps) => {
    const { motto } = stateToProps;
    const { onCreateMotto } = dispatchToProps;
    return Object.assign({}, stateToProps, dispatchToProps, ownP
rops, {
      onCreateMotto: onCreateMotto(motto),
    });
  },
)(InputModal);
```

以下是 `src/containers/MottoListContainer/MottoListContainer.js` ：

```
import { connect } from 'react-redux';
import MottoList from '../../components/MottoList';
import Immutable from 'immutable';
import uuid from 'uuid';

import {
  createMotto,
  getMottos,
  changeMottoTitle,
} from '../../actions';

export default connect(
```

```
  (state) => ({
    mottos: state.getIn(['motto', 'mottos']),
  }),
  (dispatch) => ({
    onCreateMotto: () => (
      dispatch(createMotto())
    ),
    onGetMottos: (mottos) => (
      dispatch(getMottos({ mottos }))
    ),
    onChangeMottoTitle: (title) => (
      dispatch(changeMottoTitle({ value: title }))
    ),
    // 判断点击的是哪一个 item 取出其 key，通过 itemsRef 将其移除
    onDeleteMotto: (mottos) => (id, itemsRef) => () => {
      mottos.forEach((value, key) => {
        if(value.get('id') === id) {
          itemsRef.child(key).remove();
        }
      });
    }
  }),
  (stateToProps, dispatchToProps, ownProps) => {
    const { mottos } = stateToProps;
    const { onDeleteMotto } = dispatchToProps;
    return Object.assign({}, stateToProps, dispatchToProps, ownP
rops, {
      onDeleteMotto: onDeleteMotto(mottos),
    });
  }
)(MottoList);
```

最后我们可以通过启动模拟器后使用以下命令开启我们 App！

```
$ react-native run-android
```

最后的成果：

Startup Mottos

Life is short Delete

Get Shit Done! Delete

Stay focus, keep shipping Delete

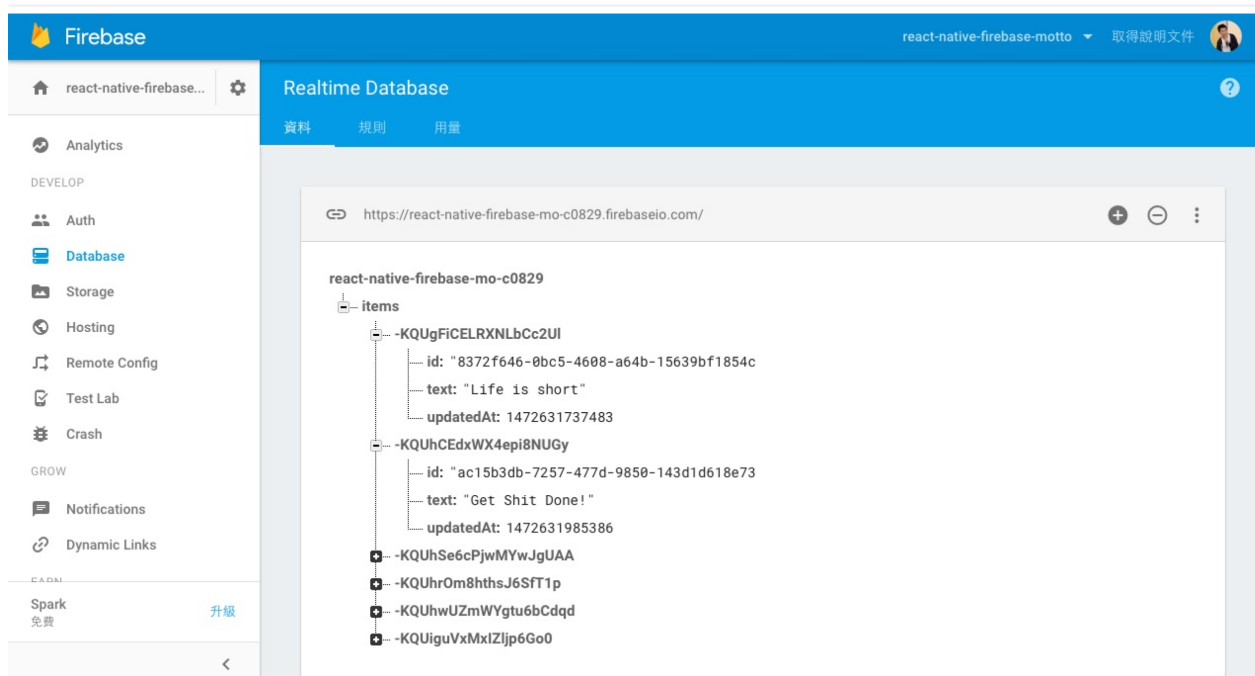
Move Fast & Break Things Delete

Its not innovation until it gets built Delete

Be Lean ! Delete

Add Motto

同时你可以在 Firebase 后台进行观察，当呼叫 Firebase API 进行数据更改时，Firebase Realtime Database 就会即时更新：



总结

恭喜你！你已经完成了你的第一个 React Native App，若你希望将你开发的应用程序审核后上架，请参考官方的说明文件，当你完成审核打包等流程后，我们可以获得 .apk 档，这时就可以上架到app store让隔壁班的女生心仪，啊不是，是广大的 Android 使用者使用你的 App 啦！当然，由于我们的代码可以 100% 共用于 iOS 和 Android 端，所以你也可以同步上架到 Apple Store！

延伸阅读

1. React Native 官方网站
2. React 官方网站
3. Redux 官方文件
4. Ionic Framework vs React Native
5. How to Build a Todo App Using React, Redux, and Immutable.js
6. Your First Immutable React & Redux App
7. React, Redux and Immutable.js: Ingredients for Efficient Web Applications
8. Full-Stack Redux Tutorial
9. redux与immutable实例
10. gajus/redux-immutable
11. acdlite/redux-actions
12. Flux Standard Action

13. React Native ImmutableJS ListView Example
14. React Native 0.23.1 warning: 'In next release empty section headers will be rendered'
15. js.coach
16. React Native Package Manager
17. React Native 学习笔记
18. The beginners guide to React Native and Firebase
19. Authentication in React Native with Firebase
20. bruz/react-native-redux-groceries
21. Building a Simple ToDo App With React Native and Firebase
22. Firebase Permission Denied
23. Best Practices: Arrays in Firebase
24. Avoiding plaintext passwords in gradle
25. Generating Signed APK

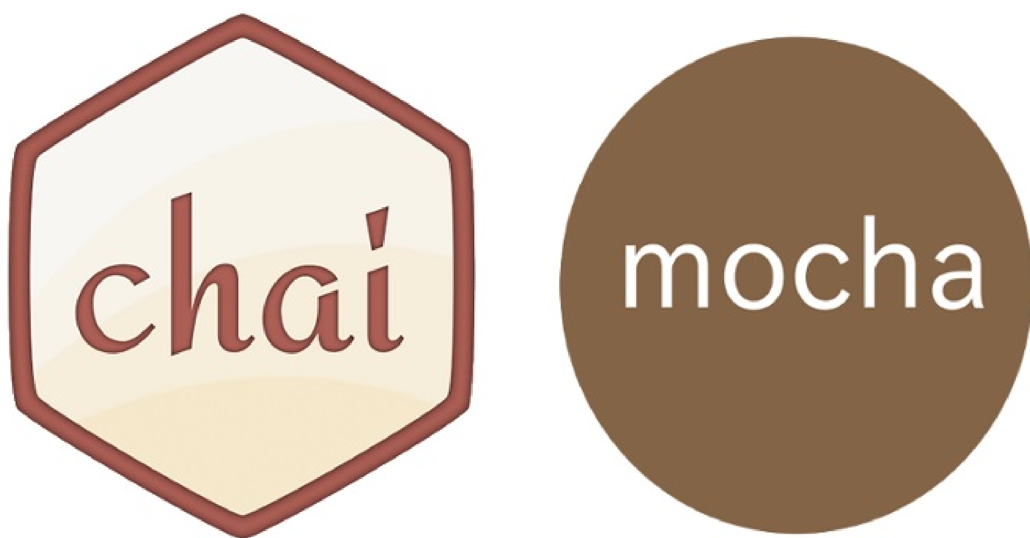
(image via moduscreate、css-tricks、teamtreehouse、teamtreehouse、css-tricks、css-tricks)

:door: 任意门

| 回首页 | 上一章：附录一、React ES5、ES6+ 常见用法对照表 | 下一章：附录三、React 测试入门教学 |

| 纠错、提问或许愿 |

附录三、React 测试入门教学



前言

测试是软体开发中非常重要的一个环节，本章我们将带领大家从编写最简单的测试代码到整合 `Mocha` + `Chai` 官方提供的测试工具和 Airbnb 所设计的 Enzyme 进行 React 测试。

Mocha 测试初体验

Mocha 是目前颇为流行的 JavaScript 测试框架之一，其可以很方便使用于浏览器端和 Node 环境。

> Mocha is a feature-rich JavaScript test framework running on Node.js and in the browser, making asynchronous testing simple and fun. Mocha tests run serially, allowing for flexible and accurate reporting, while mapping uncaught exceptions to the correct test cases.

除了 Mocha 外，尚有许多 JavaScript 单元测试工具可以选择，例如：Jasmine、Karma 等。但本章我们主要使用 `Mocha` + `Chai` 结合 React 官方测试工具和 Enzyme 进行讲解。

在这边我们先介绍一些比较常用的 Mocha 使用方法，让大家熟悉测试的用法（若是已经熟悉编写测试代码的读者这部份可以跳过）：

1. 安装环境与套件

   安装 `react` 和 `react-dom`

   ```
   $ npm install --save react react-dom
   ```

   可以在全域安装 mocha：

   ```
   $ npm install --global mocha
   ```

   也可以在开发环境下本地端安装（同时安装了 babel、eslint、webpack 等相关套件，其中以 mocha、chai、babel 为主要必须）：

   ```
   $ npm install --save-dev babel-core babel-loader babel-eslint babel-preset-react babel-preset-es2015 eslint eslint-config-airbnb eslint-loader eslint-plugin-import eslint-plugin-jsx-a11y eslint-plugin-react webpack webpack-dev-server html-webpack-plugin chai mocha
   ```

2. 测试代码

   i. describe（test suite）：表示一组相关的测试。 `describe` 为一个函数，第一个参数为 `test suite` 的名称，第二个参数为实际执行的函数。
   ii. it（test case）：表示一个单独测试，为测试里最小单位。 `it` 为一个函数，第一个参数为 `test case` 的描述名称，第二个参数为实际执行的函数。

      在测试代码中会包含一个或多个 `test suite`，而每个 `test suite` 则会包含一个或多个 `test case`。

3. 整合 assertion 函数库 `Chai`

   所谓的 assertion（断言），就是判断代码的执行成果是否和预期一样，若是不一致则会发生错误。通常一个 test case 会拥有一个或多个 assertion。由于 Mocha 本身是一个测试框架，但不包含 assertion，所以我们使用 Chai 这个适

用于浏览器端和 Node 端的 BDD / TDD assertion library。在 Chai 中共提供三种操作 assertion 介面风格：Expect、Assert、Should，在这边我们选择使用比较接近自然语言的 Expect。

基本上，expect assertion 的写法都是类似：开头为 `expect` 方法 + `to` 或 `to.be` + 结尾 assertion 方法（例如：equal、a/an、ok、match）

4. Mocha 基本用法

mocha 若没指定要执行哪个文件，预设会执行 `test` 文件夹下第一层的测试代码。若要让 `test` 文件夹中的子文件夹测试码也执行则要加上 `--recursive` 参数。

包含子文件夹：

```
$ mocha --recursive
```

指定一个文件

```
$ mocha file1.js
```

也可以指定多个文件

```
$ mocha file1.js file2.js
```

现在，我们来编写一个简单的测试程序，亲身感受一下测试的感觉。以下是 `react-mocha-test-example/src/modules/add.js`，一个加法的函数：

```
const add = (x, y) => (
  x + y
);

export default add;
```

接著我们编写测试这个函数的代码，测试是否正确。以下是 `react-mocha-test-example/src/test/add.test.js`：

```
// test add.js
import add from '../src/modules/add';
import { expect } from 'chai';

// describe is test suite, it is test case
describe('test add function', () => (
  it('1 + 1 = 2', () => (
    expect(add(1, 1)).to.be.equal(2)
  ))
));
```

在开始执行 `mocha` 后由于我们使用了，ES6 的语法所以必须使用 bable 进行转译，否则会出现类似以下的错误：

```
import add from '../src/modules/add';
^^^^^^
```

我们先行设定 `.bablerc`，我们在之前已经有安装 `babel` 相关套件和 `presets` 所以就会将 ES2015 语法转译。

```
{
    "presets": [
      "es2015",
      "react",
     ],
    "plugins": []
}
```

此时，我们更改 `package.json` 中的 `scripts`，这样方便每次测试执行：

若是使用本地端：

```
$ ./node_modules/mocha/bin/mocha --compilers js:babel-core/
register
```

若是使用全域：

```
$ mocha --compilers js:babel-core/register
```

若是一切顺利，我们就可以看到执行测试成功的结果：

``` $ mocha add.test.js

test add function

```
✓ 1 + 1 = 2
```

```
  1 passing (181ms)
```
```

1. Mocha 指令参数

   在 Mocha 中有许多可以使用的好用参数，例如：`--recursive` 可以执行执行测试文件夹下的子文件夹代码、`--reporter 格式` 更改测试报告格式（预设是 `spec`，也可以更改为 `tap`）、`--watch` 用来监控测试代码，当有测试代码更新就会重新执行、`--grep` 撷取符合条件的 test case。

   以上这些参数我们可以都整理在 `test` 文件夹下的 `mocha.opts` 文件中当作设定数据，此时再次执行 `npm run test` 就会把参数也使用进去。

   ```
   --watch
   --reporter spec
   ```

2. 非同步测试

   在上面我们讨论的主要是同步的状况，但实际上在开发应用时往往会遇到非同步的情形。而在 Mocha 中每个 test case 最多允许执行 2000 毫秒，当时间超过就会显示错误。为了解决这个问题我们可以在 `package.json` 中更改：`"test": "mocha -t 5000 --compilers js:babel-core/register"` 文件。

   为了模拟测试非同步的情境，所以我们必须先安装 axios。

```
$ npm install --save axios
```

以下是 `react-mocha-test-example/src/test/async.test.js` ：

```
import axios from 'axios';
import { expect } from 'chai';

it('asynchronous return an object', function(done){
  axios
    .get('https://api.github.com/users/torvus')
    .then(function (response) {
      expect(response).to.be.an('object');
      done();
    })
    .catch(function (error) {
      console.log(error);
    });
});
```

由于测试环境是在 Node 中，所以我们必须先安装 node-fetch 来展现 promise 的情境。

```
$ npm install --save node-fetch
```

以下是 `react-mocha-test-example/src/test/promise.test.js` ：

```
import fetch from 'node-fetch';
import { expect } from 'chai';

it('asynchronous fetch promise', function() {
  return fetch('https://api.github.com/users/torvus')
    .then(function(response) { return response.json() })
    .then(function(json) {
      expect(json).to.be.an('object');
    });
});
```

3. 测试使用的 hook

   在 Mocha 中的 test suite 中，有 before()、after()、beforeEach() 和 afterEach() 四种 hook，可以让你设计在特定时间点执行测试。

```
describe('hooks', function() {
  before(function() {
    // 在 before 中的 test case 会在所有 test cases 前执行
  });
  after(function() {
    // 在 after 中的 test case 会在所有 test cases 后执行
  });
  beforeEach(function() {
    // 在 beforeEach 中的 test case 会在每个 test cases 前执行
  });
  afterEach(function() {
    // 在 afterEach 中的 test case 会在每个 test cases 后执行
  });
  // test cases
});
```

动手实践

在上面我们已经先讲解了 `Mocha` + `Chai` 测试工具和基础的测试写法。现在接著我们要来探讨 React 中的测试用法。然而，要在 React 中测试 Component 以及 JSX 语法时，使用传统的测试工具并不方便，所以要整合 `Mocha` + `Chai` 官方提供的测试工具和 Airbnb 所设计的 Enzyme（由于官方的测试工具使用起来不太方便所以有第三方针对其进行封装）进行测试。

使用官方测试工具

我们知道在 React 一个重要的特色为 Virtual DOM 所以在官方的测试工具中有提供测试 Virtual DOM 的方法：Shallow Rendering（createRenderer），以及测试真实 DOM 的方法：DOM Rendering（renderIntoDocument）。

1. Shallow Rendering（createRenderer）

Shallow Rendering 系指将一个 Virtual DOM 渲染成子 Component，但是只渲染第一层，不渲染所有子组件，因此处理速度快且不需要 DOM 环境。Shallow rendering 在单元测试非常有用，由于只测试一个特定的 component，而重要的不是它的 children。这也意味著改变一个 child component 不会影响 parent component 的测试。

以下是 `react-addons-test-utils-example/src/test/shallowRender.test.js`：

```
import React from 'react';
import TestUtils from 'react-addons-test-utils';
import { expect } from 'chai';
import Main from '../src/components/Main';

function shallowRender(Component) {
  const renderer = TestUtils.createRenderer();
  renderer.render(<Component/>);
  return renderer.getRenderOutput();
}

describe('Shallow Rendering', function () {
  it('Main title should be h1', function () {
    const todoItem = shallowRender(Main);
    expect(todoItem.props.children[0].type).to.equal('h1');
    expect(todoItem.props.children[0].props.children).to.eq
ual('Todos');
  });
});
```

以下是 `react-addons-test-utils-example/src/test/shallowRenderProps.test.js`：

```
import React from 'react';
import TestUtils from 'react-addons-test-utils';
import { expect } from 'chai';
import TodoList from '../src/components/TodoList';

const shallowRender = (Component, props) => {
  const renderer = TestUtils.createRenderer();
  renderer.render(<Component {...props}/>);
  return renderer.getRenderOutput();
}

describe('Shallow Props Rendering', () => {
  it('TodoList props check', () => {
    const todos = [{ id: 0, text: 'reading'}, { id: 1, text
: 'coding'}];
    const todoList = shallowRender(TodoList, {todos: todos}
);
    expect(todoList.props.children.type).to.equal('ul');
    expect(todoList.props.children.props.children[0].props.
children).to.equal('reading');
    expect(todoList.props.children.props.children[1].props.
children).to.equal('coding');
  });
});
```

2. DOM Rendering（renderIntoDocument）

注意，因为 Mocha 运行在 Node 环境中，所以你不会存取到 DOM。所以我们要使用 JSDOM 来模拟真实 DOM 环境。同时我在这边引入 `react-dom` ，这样我们就可以使用 findDOMNode 来选取元素。事实上，findDOMNode 方法的最大优势是提供比 TestUtils 更好的 CSS 选择器，方便开发者选择元素。

以下是 `react-addons-test-utils-example/src/test/setup.test.js` ：

```
import jsdom from 'jsdom';

if (typeof document === 'undefined') {
  global.document = jsdom.jsdom('<!doctype html><html><head
></head><body></body></html>');
  global.window = document.defaultView;
  global.navigator = global.window.navigator;
}
```

以下是 `react-addons-test-utils-example/src/components/TodoHeader/TodoHeader.js`：

```
import React from 'react';

class TodoHeader extends React.Component {
  constructor(props) {
    super(props);
    this.toggleButton = this.toggleButton.bind(this);
    this.state = {
      isActivated: false,
    };
  }
  toggleButton() {
    this.setState({
      isActivated: !this.state.isActivated,
    })
  }
  render() {
    return (
      <div>
        <button disabled={this.state.isActivated} onClick={
this.toggleButton}>Add</button>
      </div>
    );
  };
}

export default TodoHeader;
```

需要留意的是若是 stateless components 使用 TestUtils.renderIntoDocument，要将 renderIntoDocument 包在 `<div></div>` 内，使用 `findDOMNode(TodoHeaderApp).children[0]` 取得，不然会回传 null。更进一步细节可以参考这里。不过由于我们是使用 `class-based` Component 所以不会遇到这个问题。

以下是 `react-addons-test-utils-example/src/test/renderIntoDocument.test.js`：

```
import React from 'react';
import TestUtils from 'react-addons-test-utils';
import { expect } from 'chai';
import { findDOMNode } from 'react-dom';
import TodoHeader from '../src/components/TodoHeader';

describe('Simulate Event', function () {
  it('When click the button, it will be toggle', function () {
    const TodoHeaderApp = TestUtils.renderIntoDocument(<TodoHeader />);
    const TodoHeaderDOM = findDOMNode(TodoHeaderApp);
    const button = TodoHeaderDOM.querySelector('button');
    TestUtils.Simulate.click(button);
    let todoHeaderButtonAfterClick = TodoHeaderDOM.querySelector('button').disabled;
    expect(todoHeaderButtonAfterClick).to.equal(true);
  });
});
```

这种渲染 DOM 的测试方式类似于 JavaScript 或 jQuery 的 DOM 操作。首先要先找到要想操作的目标节点，而后触发想要执行的动作，在官方测试工具中拥有许多可以协助选取节点的方法。然而由于其在使用上不够简洁，也因此我们接下来将介绍由 Airbnb 所设计的 Enzyme进行 React 测试。

使用 Enzyme 函数库进行测试

Enzyme 优势是在于针对官方测试工具封装成了类似 jQuery API 的选取元素的方式。根据官方网站介绍 Enzyme 将更容易地去操作选取 React Component：

> Enzyme is a JavaScript Testing utility for React that makes it easier to assert, manipulate, and traverse your React Components' output. Enzyme is unopinionated regarding which test runner or assertion library you use, and should be compatible with all major test runners and assertion libraries out there.

在 Enzyme 中选取元素使用 `find()` ：

```
component.find('.className'); // 使用 class 选取
component.find('#idName'); // 使用 id 选取
component.find('h1'); // 使用元素选取
```

接下来我们介绍 Enzyme 三个主要的 API 方法：

1. Shallow Rendering

   shallow 方法事实上就是官方测试工具的 shallow rendering 封装。同样是只渲染第一层，不渲染所有子组件。

   ```
   import React from 'react';
   import TestUtils from 'react-addons-test-utils';
   import { expect } from 'chai';
   import { shallow } from 'enzyme';
   import Main from '../../src/components/Main';

   describe('Enzyme Shallow Rendering', () => {
     it('Main title should be Todos', () => {
       const main = shallow(<Main />);
       // 判断 h1 文字是否如预期
       expect(main.find('h1').text()).to.equal('Todos');
     });
   });
   ```

2. Static Rendering

   render 方法是将 React 组件渲染成静态的 HTML 字串，并利用 Cheerio 函数库（这点和 shallow 不同）分析其结构返回对象。虽然底层是不同的处理引擎但使用上 API 封装起来和 Shallow 却是一致的。需要注意的是 Static Rendering 非只渲染一层，需要注意是否需要 mock props 传递。

```
import React from 'react';
import TestUtils from 'react-addons-test-utils';
import { expect } from 'chai';
import { render } from 'enzyme';
import Main from '../../src/components/Main';

describe('Enzyme Staic Rendering', () => {
  it('Main title should be Todos', () => {
    const todos = [{ id: 0, text: 'reading'}, { id: 1, text
: 'coding'}];
    const main = render(<Main todos={todos} />);
    expect(main.find('h1').text()).to.equal('Todos');
  });
});
```

3. Full Rendering

mount 方法 React 组件载入真实 DOM 节点。同样因为牵涉到 DOM 也要使用 JSDOM。

```
import React from 'react';
import TestUtils from 'react-addons-test-utils';
import { expect } from 'chai';
import { findDOMNode } from 'react-dom';
import { mount } from 'enzyme';
import TodoHeader from '../../src/components/TodoHeader';

describe('Enzyme Mount', () => {
  it('Click Button', () => {
    let todoHeaderDOM = mount(<TodoHeader />);
    // 取得 button 并模拟 click
    let button = todoHeaderDOM.find('button').at(0);
    button.simulate('click');
    // 检查 prop(key) 是否正确
    expect(button.prop('disabled')).to.equal(true);
  });
});
```

最后我们可以在 `react-addons-test-utils-example` 文件夹下执行：

```
$ npm test
```

若一切顺利就可以看到测试通过的消息！

```
  Enzyme Mount
    ✓ Click Button (44ms)

  Enzyme Shallow Rendering
    ✓ Main title should be Todos

  Enzyme Staic Rendering
    ✓ Main title should be Todos

  Simulate Event
    ✓ When click the button, it will be toggle

  Shallow Rendering
    ✓ Main title should be h1

  Shallow Props Rendering
    ✓ TodoList props check


  6 passing (279ms)
```

事实上 Enzyme 还提供更多的 API 可以使用，若是读者想了解更多 Enzyme API 可以 参考官方文件。

总结

以上我们从 `Mocha` + `Chai` 的使用方式介绍到 React 官方提供的测试工具 和 Airbnb 所设计的 Enzyme，相信读者对于测试代码已经有初步的了解，若尚未掌握的读者不妨跟著上面的范例再重新走过一遍，接著我们要进到最后的 `GraphQL/Relay` 的介绍。

延伸阅读

1. React 测试入门教程
2. 测试框架 Mocha 实例教程
3. Test Utilities
4. JavaScript Testing utilities for React
5. 持续集成是什么？
6. Let’s test React components with TDD, Mocha, Chai, and jsdom
7. Unit Testing React-Native Components with Enzyme Part 1
8. What React Stateless Components Are Missing
9. 0.14-rc1: findDOMNode(statelessComponent) doesn’t work with TestUtils.renderIntoDocument #4839
10. Writing Redux Tests
11. 【译】展望2016，React.js 最佳实践 (中英对照版)

（image via Anthony Ng）

:door: 任意门

| 回首页 | 上一章：附录二、用 React Native + Firebase 开发跨平台行动应用程序 | 下一章：附录四、GraphQL/Relay 初体验 |

| 纠错、提问或许愿 |

附录四、GraphQL/Relay 初体验



前言

GraphQL 的出现主要是为了要解决 Web/Mobile 端不断增加的 API 请求所衍生的问题。由于 RESTful 最大的功能在于很有效的前后端分离和建立 stateless 请求，然而 RESTful API 的资源设计上比较偏向单方面的互动，若是有著复杂资源间的关联就会出现请求次数过多，遇到不少的瓶颈。

GraphQL 初体验

> GraphQL is a data query language and runtime designed and used at Facebook to request and deliver data to mobile and web apps since 2012.

根据 GraphQL 官方网站的定义，GraphQL 是一个数据查询语言和 runtime。Query responses 是由 client 所定义决定，而非 server 端，且只会回传 client 所定义的内容。此外，GraphQL 是强型别（strong type）且可以容易使用阶层（hierarchical）和处理复杂的数据关连性，并更容易让前端工程师和产品工程师定义 Schema 来使用，赋予前端对于数据的制定能力。

GraphQL 主要由以下组件构成：

1. 类别系统（Type System）
2. 查询语言（Query Language）：在 Operations 中 query 只读取数据而 mutation 写入操作
3. 执行语意（Execution Semantics）
4. 静态验证（Static Validation）
5. 类别检查（Type Introspection）

一般 RESTful 在取用资源时会对应到 HTTP 中 `GET`、`POST`、`DELETE`、`PUT` 等方法，并以 URL 对应的方式去取得资源，例如：

取得 id 为 3500401 的使用者数据：

GET `/users/3500401`

以下则是 GraphQL 定义的 query 范例，定义式（declarative）的方式比起 RESTful 感觉起来相对直观：

```
{
  user(id: 3500401) {
    id,
    name,
    isViewerFriend,
    profilePicture(size: 50)  {
      uri,
      width,
      height
    }
  }
}
```

接收到 GraphQL query 后 server 回传结果：

```
{
  "user" : {
    "id": 3500401,
    "name": "Jing Chen",
    "isViewerFriend": true,
    "profilePicture": {
      "uri": "http://someurl.cdn/pic.jpg",
      "width": 50,
      "height": 50
    }
  }
}
```

实战演练

在 GraphQL 中有取得数据 Query、更改数据 Mutation 等操作。以下我们先介绍如何建立 GraphQL Server 并取得数据。

1. 环境建置 接下来我们将动手建立 GraphQL 的简单范例，让大家感受一下 GraphQL 的特性，在这之前我们需要先安装以下套件建立好环境：

    i. graphql：GraphQL 的 JavaScript 实践.
    ii. express：Node web framework.
    iii. express-graphql, an express middleware that exposes a GraphQL server.

    ```
    $ npm init
    $ npm install graphql express express-graphql --save
    ```

2. Data 格式设计

    以下是 `data.json` ：

```
{
  "1": {
    "id": "1",
    "name": "Dan"
  },
  "2": {
    "id": "2",
    "name": "Marie"
  },
  "3": {
    "id": "3",
    "name": "Jessie"
  }
}
```

3. Server 设计

```
// 引入函数库
import graphql from 'graphql';
import graphqlHTTP from 'express-graphql';
import express from 'express';

// 引入 data
const data = require('./data.json');

// 定义 User type 的两个子 fields：`id` 和 `name` 字串，注意型别
对于 GraphQL 非常重要
const userType = new graphql.GraphQLObjectType({
  name: 'User',
  fields: {
    id: { type: graphql.GraphQLString },
    name: { type: graphql.GraphQLString },
  }
});

const schema = new graphql.GraphQLSchema({
  query: new graphql.GraphQLObjectType({
    name: 'Query',
    fields: {
```

```
        user: {
          // 使用上面定义的 userType
          type: userType,
          // 定义所接受的 user 参数
          args: {
            id: { type: graphql.GraphQLString }
          },
          // 当传入参数后 resolve 如何处理回传 data
          resolve: function (_, args) {
            return data[args.id];
          }
        }
      }
    })
  });

  // 启动 graphql server
  express()
    .use('/graphql', graphqlHTTP({ schema: schema, pretty: true }))
    .listen(3000);

  console.log('GraphQL server running on http://localhost:3000/graphql');
```

在终端机执行：

```
node index.js
```

这个时候我们可以打开浏览器输入 `localhost:3000/graphql.` ，由于没有任何 Query，目前会出现以下画面：

```
1   // 20160908211131
2   // http://localhost:3000/graphql?query={user(id:%221%22){name}}
3
4   {
5     "data": {
6       "user": {
7         "name": "Dan"
8       }
9     }
10  }
```

4. Query 设计

   当 GraphQL 指令为：

   ```
   {
     user(id: "1") {
       name
     }
   }
   ```

   将回传数据：

   ```
   {
     "data": {
       "user": {
         "name": "Dan"
       }
     }
   }
   ```

   在了解了数据和 Query 设计后，这个时候我们可以打开浏览器输入（当然也可以通过终端机 curl 的方式执行）：`http://localhost:3000/graphql?query={user(id:"1"){name}}`，此时 server 会根据 GET 的数据回传：

```
// 20160908211840
// http://localhost:3000/graphql

{
  "errors": [
    {
      "message": "Must provide query string."
    }
  ]
}
```

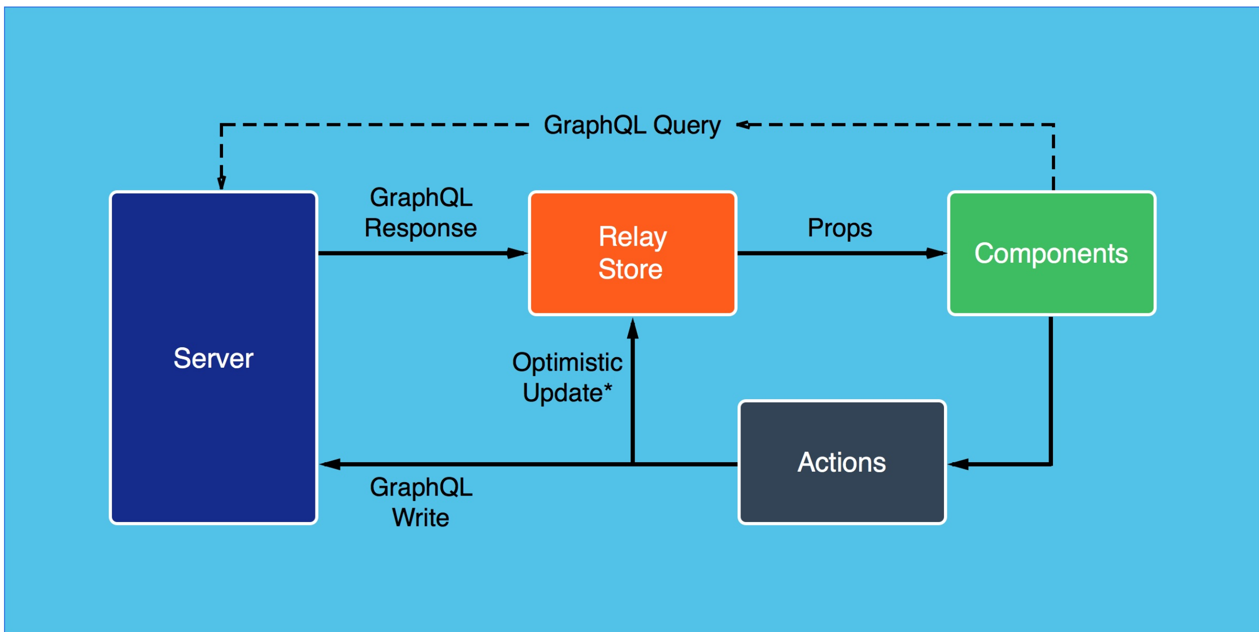
到这里，你已经完成了最简单的 GraphQL Server 设计了，若你遇到编码问题，可以尝试使用 JavaScript 中的 `encodeURI` 去进行转码。也可以自己尝试不同的 Schema 和 Query，感受一下 GraphQL 的特性。事实上，GraphQL 还拥有许多有趣的特色，例如：Fragment、指令、Promise 等，若读者对于 GraphQL 有兴趣可以进一步参考 GraphQL 官网。

Relay 初体验

> Relay is a new framework from Facebook that provides data-fetching functionality for React applications.

在体验完 GraphQL 后，我们要来聊聊 Relay。Relay 是 Facebook 为了满足大型应用程序开发所建构的框架，主要用于处理 React 应用层（Application）的数据互动框架。在 Relay 中可以让每个 Component 通过 GraphQL 的整合处理可以精确地向 Component props 提供取得的数据，并在 client side 存放一份所有数据的 store 当作暂存。

整个 Relay 架构流程图：



一般来说要使用 Relay 必须先准备好以下三项工具：

1. A GraphQL Schema
    - graphql-js
    - graphql-relay-js
2. A GraphQL Server
    - express
    - express-graphql
3. Relay
    - network layer：Relay 通过 network layer 传 GraphQL 给 server

接下来我们来通过 React 官方上的范例来让大家感受一下 Relay 的特性。上面我们有提过：在 Relay 中可以让每个 Component 通过 GraphQL 的整合处理可以更精确地向 Component props 提供取得的数据，并在 client side 存放一份所有数据的 store 当作暂存。所以，首先我们先建立每个 Component 和 GraphQL/Relay 的对应：

```
// 建立 Tea Component，从 this.props.tea 取得数据
class Tea extends React.Component {
  render() {
    var {name, steepingTime} = this.props.tea;
    return (
      <li key={name}>
        {name} (<em>{steepingTime} min</em>)
```

```
        </li>
      );
    }
  }
  // 使用 Relay.createContainer 建立数据沟通窗口
  Tea = Relay.createContainer(Tea, {
    fragments: {
      tea: () => Relay.QL`
        fragment on Tea {
          name,
          steepingTime,
        }
      `,
    },
  });

  class TeaStore extends React.Component {
    render() {
      return <ul>
        {this.props.store.teas.map(
          tea => <Tea tea={tea} />
        )}
      </ul>;
    }
  }
  TeaStore = Relay.createContainer(TeaStore, {
    fragments: {
      store: () => Relay.QL`
        fragment on Store {
          teas { ${Tea.getFragment('tea')} },
        }
      `,
    },
  });

  // Route 设计
  class TeaHomeRoute extends Relay.Route {
    static routeName = 'Home';
    static queries = {
      store: (Component) => Relay.QL`
```

```
      query TeaStoreQuery {
        store { ${Component.getFragment('store')} },
      }
     `,
  };
}

ReactDOM.render(
  <Relay.RootContainer
    Component={TeaStore}
    route={new TeaHomeRoute()}
  />,
  mountNode
);
```

GraphQL Schema 和 store 建立：

```
// 引入函数库
import {
  GraphQLInt,
  GraphQLList,
  GraphQLObjectType,
  GraphQLSchema,
  GraphQLString,
} from 'graphql';

// client side 暂存 store，GraphQL Server reponse 会更新 store，再通过 props 传递给 Component
const STORE = {
  teas: [
    {name: 'Earl Grey Blue Star', steepingTime: 5},
    {name: 'Milk Oolong', steepingTime: 3},
    {name: 'Gunpowder Golden Temple', steepingTime: 3},
    {name: 'Assam Hatimara', steepingTime: 5},
    {name: 'Bancha', steepingTime: 2},
    {name: 'Ceylon New Vithanakande', steepingTime: 5},
    {name: 'Golden Tip Yunnan', steepingTime: 5},
    {name: 'Jasmine Phoenix Pearls', steepingTime: 3},
    {name: 'Kenya Milima', steepingTime: 5},
```

```
    {name: 'Pu Erh First Grade', steepingTime: 4},
    {name: 'Sencha Makoto', steepingTime: 2},
  ],
};

// 设计 GraphQL Type
var TeaType = new GraphQLObjectType({
  name: 'Tea',
  fields: () => ({
    name: {type: GraphQLString},
    steepingTime: {type: GraphQLInt},
  }),
});

// 将 Tea 整合进来
var StoreType = new GraphQLObjectType({
  name: 'Store',
  fields: () => ({
    teas: {type: new GraphQLList(TeaType)},
  }),
});

// 输出 GraphQL Schema
export default new GraphQLSchema({
  query: new GraphQLObjectType({
    name: 'Query',
    fields: () => ({
      store: {
        type: StoreType,
        resolve: () => STORE,
      },
    }),
  }),
});
```

限于篇幅，我们只能让大家感受一下 Relay 的简单范例，若大家想进一步体验 Relay 的优势，已经帮你准备好 GraphQL Server、transpiler 的 Relay Starter Kit 项目会是个很好的开始。

总结

React 生态系中，除了前端 View 的部份有革新性的创新外，GraphQL 更是对于数据取得的全新思路。虽然 GraphQL 和 Relay 已经成为开源项目，但技术上仍持续演进，若需要在团队 production 上导入仍可以持续观察。到这边，若是一路从第一章看到这里的读者真的要给自己一个热烈掌声了，我知道对于初学者来说 React 庞大且有许多的新的观念需要消化，但如同笔者在最初时所提到的，学习 React 重要的是通过这个生态系去学习现代化网页开发的工具和方法以及思路，成为更好的开发者。根据前端摩尔定律，每半年就有一次大变革，但基本 Web 问题和观念依然不变，大家一起加油啦！若有任何问题都欢迎来信给笔者或是发 `issue` ，当然 PR is welcome :)

延伸阅读

1. Your First GraphQL Server
2. 搭建你的第一个 GraphQL 服务器
3. Learn GraphQL
4. GraphQL vs Relay
5. GraphQL 官网
6. Relay 官网
7. A reference implementation of GraphQL for JavaScript
8. 深入理解 GraphQL
9. Node.js 服务端实践之 GraphQL 初探

（image via facebook、kadira）

:door: 任意门

| 回首页 | 上一章：附录三、React 测试入门教学 |

| 纠正、提问或许愿 |